USENIX Association

Proceedings of the

2020 USENIX Annual Technical Conference

July 15–17, 2020

ISBN 978-1-939133-14-4

Cover Image created by freevector.com and distributed under the Creative Commons Attribution-ShareAlike 4.0 license (https://creativecommons.org/licenses/by-sa/4.0/).

Conference Organizers

Program Co-Chairs

Ada Gavrilovska, *Georgia Institute of Technology*

Erez Zadok, *Stony Brook University*

Program Committee Leaders

Aruna Balasubramanian, *Stony Brook University*

Donald Porter, *The University of North Carolina at Chapel Hill*

Liuba Shrira, *Brandeis University*

Swaminathan Sundararaman, *Pyxeda AI*

Vasily Tarasov, *IBM Research-Almaden*

Program Committee

Sangeetha Abdu Jyothi, *VMware Research and University of California, Irvine*

Rachit Agarwal, *Cornell University*

Nitin Agrawal, *ThoughtSpot*

Irfan Ahmad, *CachePhysics*

Gustavo Alonso, *ETH Zurich*

Deniz Altinbuken, *Google*

George Amvrosiadis, *Carnegie Mellon University*

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*

Behnaz Arzani, *Microsoft Research*

Mona Attariyan, *Google*

Anirudh Badam, *Microsoft Research*

Saurabh Bagchi, *Purdue University*

Antonio Barbalace, *Stevens Institute of Technology*

Ran Ben Basat, *Harvard University*

Orna Agmon Ben-Yehuda, *Technion—Israel Institute of Technology*

Pramod Bhatotia, *University of Edinburgh*

Ken Birman, *Cornell University*

Sergey Blagodurov, *AMD Research*

Herbert Bos, *Vrije Universiteit Amsterdam*

James Bottomley, *IBM Research*

Kevin Butler, *University of Florida*

Yinzhi Cao, *Johns Hopkins University*

Zhen Cao, *Google*

Feng Chen, *Louisiana State University*

Ming Chen, *Google*

Young-ri Choi, *Ulsan National Institute of Science and Technology (UNIST)*

Guilherme Cox, *Nvidia*

Heming Cui, *The University of Hong Kong (HKU)*

Dilma Da Silva, *Texas A&M University*

Alex Daglis, *Georgia Institute of Technology*

David Devescery, *Georgia Institute of Technology*

Abhinav Duggal, *Dell EMC*

Michael Ferdman, *Stony Brook University*

Pedro Fonseca, *Purdue University*

Moshe Gabel, *University of Toronto*

Manya Ghobadi, *Massachusetts Institute of Technology*

Jana Giceva, *Imperial College London*

Ashvin Goel, *University of Toronto*

Xiaohui (Helen) Gu, *North Carolina State University*

Vishakha Gupta-Cledat, *ApertureData*

Andreas Haeberlen, *University of Pennsylvania*

Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*

Danny Harnik, *IBM Research—Haifa*

Tim Harris, *Amazon*

Eshcar Hillel, *Yahoo Research*

Michio Honda, *University of Edinburgh*

Yu Hua, *Huazhong University of Science and Technology*

Jian Huang, *The University of Illinois at Urbana-Champaign*

Trent Jaeger, *The Pennsylvania State University*

Bill Jannen, *Williams College*

Junchen Jiang, *University of Chicago*

Changhee Jung, *Purdue University*

Sudarsun Kannan, *Rutgers University*

Baris Kasikci, *University of Michigan*

Idit Keidar, *Technion—Israel Institute of Technology*

Ana Klimovic, *Google Brain*

Michael Kozuch, *Intel Labs*

Orran Krieger, *Boston University*

Mohan Kumar Kumar, *Facebook*

Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*

Julia Lawall, *Inria/LIP6*

Philip Levis, *Stanford University*

Felix Lin, *Purdue University*

Heiner Litz, *University of California, Santa Cruz*

Vincent Liu, *University of Pennsylvania*

Brandon Lucia, *Carnegie Mellon University*

Xiaosong Ma, *Qatar Computing Research Institute (QCRI), HBKU, Qatar*

Peter Macko, *NetApp*

Harsha V. Madhyastha, *University of Michigan*

Carlos Maltzahn, *University of California, Santa Cruz*

Alexander Merritt, *Intel*

Michael Mesnier, *Intel Labs*

Changwoo Min, *Virginia Polytechnic Institute and State University*

Shuai Mu, *Stony Brook University*

Gilles Muller, *Inria*

Kiran-Kumar Muniswamy-Reddy, *Oracle*

Srinivasan Narayanamurthy, *NetApp*

Ravi Netravali, *University of California, Los Angeles*

Radhika Niranjan Mysore, *VMware Research*

Roberto Palmieri, *Lehigh University*

Aurojit Panda, *New York University*

Gabriel Parmer, *George Washington University*

Raju Rangaswami, *Florida International University*

Oriana Riva, *Microsoft Research*

Amitabha Roy, *Google*

Larry Rudolph, *Two Sigma*

Jack Sampson, *The Pennsylvania State University*

Mahadev Satyanarayanan, *Carnegie Mellon University*

Jiri Schindler, *Tranquil Data*

Malte Schwarzkopf, *Brown University*

Russell Sears, *Apple*

Siddhartha Sen, *Microsoft Research*

Philip Shilane, *Dell EMC*
Mark Silberstein, *Technion—Israel Institute of Technology*
Richard Spillane, *VMware Research*
Scott Stoller, *Stony Brook University*
Patrick Stuedi, *LinkedIn*
Lalith Suresh, *VMware Research*
Amy Tai, *VMware Research*
Eno Thereska, *Amazon*
Theodore Ts'o, *Google*
Alexey Tumanov, *Georgia Institute of Technology*
Keval Vora, *Simon Fraser University*
Ric Wheeler, *Facebook*
Avani Wildani, *Emory University*
Emmett Witchel, *The University of Texas at Austin*
Yue Yang, *Micron Technology*
Ming Zhao, *Arizona State University*
Lin Zhong, *Rice University*
Noa Zilberman, *University of Oxford*

External Review Committee

Raja Appuswamy, *EURECOM, Sophia Antipolis*
Muli Ben-Yehuda, *LightBits Labs*
Angela Demke Brown, *University of Toronto*
Sindhu Ghanta, *Pyxeda Inc.*
Gernot Heiser, *UNSW Sydney*
Paul E. McKenney, *IBM Linux Technology Center*
Ethan Miller, *University of California, Santa Cruz, and Pure Storage*
Dalit Naor, *IBM Research—Haifa*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
Gala Yadgar, *Technion—Israel Institute of Technology*

Submissions Co-Chairs

Ketan Bhardwaj, *Georgia Institute of Technology*
Dongyoon Lee, *Stony Brook University*

External Reviewers

Amogh Akshintala
Ali Anwar
Avi Asayag
Gal Assa
Jia-Ju Bai
Lauren Bindschaedler
Yuval Cassuto
Joanna Che
Yitao Chen
Peter Chubb
Shir Cohen
Dave Dice
Babak Esmaeili
Sungho Hong
Trammell Hudson
Kasra Jamshidi
Runyu Jin
Geoff Kuenning
Eugene Kuznetsov
Yubo Liu
Mugilan Mariappan
Matthew Morrison
Oded Naor
Arik Rinberg
Donour Sizemore
Weijia Song
Lynus Vaz
Tianzheng Wang
David Yakira
Qirui Yang
Yiting Yao
Ding Yuan
Kaiqi Zhao

2020 USENIX Annual Technical Conference

July 15–17, 2020

Wednesday, July 15

The Non-Volatile One

The Data Center One

The Cloudy One

The Buggy One

Thursday, July 16

The Machine Learning One

The OS and Virtualization One

The WAN One

The One about Big Data

Friday, July 17

The One about Acceleration

The One about Storage

The Memorable One

The One on the Edge

Message from the USENIX ATC ’20 Program Co-Chairs

1. Preamble

Every year, conference chairs share their thoughts about the conference. This is often chock-full of statistics about acceptance rates, etc. We’ll include stats for sure. We felt, however, that we’d like to share with all of you some of the “inside scoop” of how a conference is organized and run from the Co-Chairs’ perspective. Many authors don’t know what happens behind the scenes; and even most PC members don’t get to chair a conference of this magnitude. We are honored to have been selected to co-chair USENIX ATC ’20. Running a conference like ATC is substantially more work than some other, even well established conferences, for the reasons outlined below (and that effort skyrocketed when the pandemic hit). There is a lot to know and we hope that this information will be valuable to all of you in the future.

2. Early Decisions

ATC is a large and complex conference to run for several reasons.

First, while it is a systems conference, ATC is fairly broad. Whereas conferences such as OSDI, FAST, NSDI, etc. see more focused papers, ATC has a much broader set of papers that span many systems conferences: networking, storage, operating systems, security, etc. This breadth meant that the PC’s makeup had to be carefully balanced to ensure appropriate representation for all likely topics.

Second, ATC is also the place where new systems papers are often submitted for topics that may not have a home of their own. FAST, NSDI, USENIX Security, and others—all started in part because ATC was getting too many papers in those areas, suggesting it was time to form a new, more focused conference. In recent years ATC started getting papers on edge computing and applied machine learning; and this year we expected we might get for the first time ever papers in quantum computing (and we were right). This meant that the PC’s makeup also had to include people who could review papers in newer and emerging topics such as machine learning and quantum computing.

Third, ATC receives submissions from many first time or relatively junior authors. Everyone has to start somewhere. But this meant that the PC had to do more work in reviewing and shepherding papers.

It would be nearly impossible to find ATC PC Chairs who are experts in all areas of systems research, including emerging ones. Recognizing this, we decided, for the first time, to recruit a few “PC Leaders” early on. We recruited five excellent PC leaders who, in addition to reviewing papers, would also help us form the PC itself. These five Leaders are experts in areas that us Co-Chairs were not as knowledgeable in, and they were instrumental in recommending and helping us pick members for this large PC.

Another important decision we made followed previous ATCs: we selected two “Submission Co-Chairs,” one each at our respective institutions: Ketan Bhardwaj (Georgia Institute of Technology) and Dongyoon Lee (Stony Brook University). They were instrumental in assisting us run the conference submissions site (HotCRP) and overall, during the review process. They helped us analyze data, collect various statistics, write scripts, monitor online discussions, helped us run the (virtual) PC, and more. We cannot imagine running this conference without them, especially when the pandemic demanded a lot more effort on everyone’s part.

2.1. Decisions we made early on: R1 rejections, topics/themes

Every chair gets a chance to redefine the process of running the conference somewhat. We looked at past conferences as well as past ATCs and decided to try a few changes.

2.1.1. Two review rounds and early rejections

ATC and other conferences have two rounds of reviewing. In the first round (R1), every paper gets a few reviews, and we decided to keep it at 3 reviews. There would then be online discussion and filtering. Those papers deemed worthy would advance to the second round of reviewing (R2) and receive about 2 more reviews. We aimed for 3 reviews in R1 and 5 in R2. There’s an “art” of sorts to deciding how many reviews to have in each round: if you have too few, less informed decisions are made. But ironically, if you have too many reviews, in many cases that does not lead to better decisions. We have seen it before:

when you assign a paper more reviews, there's sometimes less convergence to an accept/reject decision; often the result is more variance, so converging on a decision is harder. No, more reviews aren't necessarily better. Rather, PC members have to be given time to discuss their reviews and try to convince each other of the merits of one's opinions.

Traditionally, papers that don't make it to R2 are slated for rejection. But as there's a second round of reviewing and a PC meeting, there's often a month or more between when R2 reviewing begins and all rejection notices are sent.

So we decided to send rejection notices for papers in R1 who didn't make it to R2, as quickly as we could. That was almost a month before final paper decisions were released. We did this to help authors, who now have even more time to consider their papers and the reviews they received, and decide how to revise the work for an eventual resubmission.

While sending R1 rejections helped authors, it meant more work for the PC and Co-Chairs, and a tighter schedule. Before any reviews are sent out, we decided to follow past models and perform a "Review Sufficiency Check" (RSC). In this stage, a paper reviewer is assigned to read all of the reviews and to ensure that they are complete, detailed, fair, respectful, proofread, and not missing useful information. An example of what an RSC check catches is when a reviewer might say "this work was done before," during this RSC phase, we would catch this and ask the reviewer to provide more details such as an actual citation to the "done before" work.

2.1.2. Paper topics

When authors submit papers, they check several boxes to indicate what topics their paper fits into (e.g., networking, storage, operating systems). Similarly, before PC members begin to review any paper, they log in to the conference reviewing system, HotCRP (yes, pronounced hot-crap :-), and configure their expertise level in each of the topics. PC members mark their expertise on a five-point scale from "no expertise," to "neutral," to "expert." This indicates the PC member's preference in reviewing or not reviewing papers on certain topics.

Assigning hundreds of papers to over 100 PC members is a daunting task. Thankfully, HotCRP has a built-in "auto-assignment" feature. HotCRP can match papers to PC members by mutual interest in a given topic. HotCRP's initial assignment is very helpful but is not perfect, so it still needs a lot of review and tweaking by the Co-Chairs.

One trend in recent years has been to increase the number of topics and even add "cross-cutting" themes (sometimes called "aspects"). For example, instead of just saying "networking," you break it down into local-area, wide-area, mobile, protocols, data-center, etc., networking. And instead of saying "storage" you break it down to file systems, NVMs, Flash storage, network storage, KV stores, etc. And then you add orthogonal themes that can be applied to any of the finer-grained topics, for example: performance, scalability, security, availability, reliability, scheduling, etc.

There was a good reason to add more and more fine-grained topics and themes: HotCRP could do a better matching of papers to reviewers, ensuring that the best-qualified experts would get to review a paper. But there was a growing problem with the proliferation of topics and themes.

First, asking PC members and authors to mark 60+ topics and themes was a growing burden.

Second, authors don't always mark their papers correctly, often missing important topics/themes.

Third, some PC members don't want to mark their topic preference on purpose: they would like to get a "random assignment" of papers from the pool of submissions. There is a valid reason for this. A good paper should not just appeal to experts in the paper's field, but to a wider audience from the same community. So by reviewing some papers by highly technical PC members, but not necessarily experts, you ensure the selection of papers with greater appeal to the systems community.

Fourth, and most important. We found that the "cross-cutting themes" were not helping much; in fact, they were likely hurting HotCRP's initial assignment of papers to reviews. Suppose an author marks their paper with one or more topics and the theme of "performance." A person who can review a "storage performance" paper well probably won't be able to review a "network performance" paper as well. Similarly, "scheduling" in data centers is very different from I/O scheduling or network-packet scheduling. Analyzing these themes, we decided that they were not helpful and even counter-productive, and so we decided to eliminate all cross-cutting themes from the USENIX ATC '20 CFP.

Next, we tackled proliferation in the number of themes. After much discussion that involved our PC Leaders and Submission Co-Chairs, we decided to narrow the list of topics from several dozen to just 14. The reason to reduce the number of topics was to lower the burden on authors and PC members when marking their expertise; but at the same time, we wanted to ensure that we would still be able to assign papers to expert reviewers. So we chose multi-topic themes such that expertise in one topic is likely to translate to another. For example, one topic was “Distributed Systems, Clouds, Clusters, Data Centers.” The idea was that if a PC member can review a paper in “Distributed systems,” then they should also be able to review a paper on “data centers” reasonably well. No paper-to-reviewers assignment is ever perfect, but we felt that the way we chose themes and fewer of them helped reduce overall burden and still keep the reviewing quality high.

3. PC Selection Process

PC selection is a multi-dimensional optimization problem. Co-Chairs have to ensure that the PC is well balanced as much as possible across multiple dimensions (in no particular order):

1. **Junior vs. senior people:** it’s good to have some “fresh blood” and first time PC members, but also to have senior people who provide the necessary “institutional memory” and the wisdom that comes with... gray hair.
2. **Industry vs. academia:** it’s not enough to have just academics on a PC. We have to have representation from industry, who work on many real-world cutting-edge problems.
3. **Domestic vs. international:** the systems community is spread around the world. We want representation not just from U.S. researchers, but from Europe, Far and Middle East, and more.
4. **Male vs. female:** notwithstanding that genders are not binary, computer science and systems research are sorely lacking in female representation. This is true in academia (faculty and students), as well as industry. Inclusivity and diversity has proven to improve outcomes of many decision-making processes. Alas, our community has far fewer female researchers than we’d like to have. Many female researchers get asked repeatedly to serve on PCs, creating perhaps an unfair load. We tried to invite as many female PC members to join as we could: 24% of invited were female; and in the end our PC had 23% female representation.
5. **Topic diversity:** it’s important for the PC to have just the right number of people to review papers in various topics. We reviewed stats from previous years, such as how many papers were received in traditional topics (e.g., storage, clouds) as well as emerging topics (e.g., edge, ML). We estimated how many papers we might get overall and how many papers in each topic. We planned for getting anywhere from 300 to 500 (worst case) submissions. In the end, we had just under 350 reviewed submissions. For each PC member we considered inviting, we had a spreadsheet to mark their expertise in various areas.

Next, we chose to follow previous year’s models and invite PC members in multiple rounds and for multiple effort levels. During the fall of 2019 we sent out eight different rounds of invitations. After waiting to hear back, we reviewed the current makeup of the PC along the five aforementioned dimensions. We used google sheets and docs extensively. We then adjusted our strategy and sent out the next round of invitations, consulting with our PC Leaders as necessary. Adjustment was necessary to ensure that, if we were short in one dimension above, that we doubled our efforts to catch up along that dimension without hurting other dimensions.

We offered three tiers of reviewing load. Heavy PC members were expected to review 16–20 papers; Light PC members were expected to review 8–10 papers; and External Review Committee (ERC) members were expected to review 1–3 papers at most. Heavy PC members were required to attend the physical PC meeting (which was converted to a virtual PC—more on that later). ERC members were selected for specialized expertise in narrow areas where we felt we would need a couple more reviews. All PC members were expected to review papers in two rounds and participate in both rounds’ online discussions. We first invited Heavy, then Light, then ERC. In some cases, those who declined our invitation to be Heavy PC members were offered to be a Light PC member (several agreed).

We made a total of 214 PC invitations. 42% of all invited declined (many are understandably busy). Of the 125 that accepted, we had 67 Heavy PC members, 48 Light, and 10 ERC.

CFP updated and posted, PC formed, HotCRP configured, we were ready...

... or so we thought.

> *Con permiso, Capitán. The hall is rented, the orchestra engaged. It’s now time to see if you can dance.*
>
> –Q to Captain Picard, “Q Who,” Star Trek: The Next Generation

4. Before Paper Submission Due Date

Anonymizing papers helps ensure the integrity of the conference. When neither authors nor PC members know each other's identity, this is called a double-blind review process.

We received dozens of emails from authors who asked for clarifications on how to anonymize their papers. This is always a difficult subject. If you anonymize your paper too much, you may be hiding information about useful prior work you did, which could help the current paper under submission. Ironically, if you anonymize your past work too well, you could be incorrectly accused of reproducing or even plagiarizing "someone else's work" where it's really your own work.

If you don't anonymize, or poorly anonymize, you expose your identity, even indirectly to PC members. There's no consensus whether it helps or hurts a paper's chances, but there is consensus that it biases the process one way or another.

Many ATC authors are first-timers or relatively junior, which explains why we received many such queries about the anonymization process. Our advice to authors was as follows. PC members are encouraged not to search the Internet for information that might reveal the authors' identities. So authors had to ensure that their papers were anonymized and self-contained. Citations to one's own paper should be stated in the third person: you can refer to your own past work as "John Doe et al. [1] did this or that" and then state how you improved on those "other" people's work.

There are times when anonymization can get very difficult. Suppose the authors are from a very large cloud operator and are reporting on their experiences running a million computers across dozens of data centers worldwide. This kind of experience is invaluable to report to the systems community, as most researches can only dream of having access to such a large set of computers and its associated data sets. Therefore it doesn't make sense in such a paper, to "hide" details as to the size and scope of the environment being studied: hiding such details would likely hurt such a paper considerably. However, anyone reviewing such an anonymized paper would have a pretty good guess which company the authors work for. After all, we can all count on one hand the number of cloud operators with such a large operation.

The above was just one example. We generally told authors that anonymizing is a "best-effort" process: try your best to hide your identity but avoid going too far that it genuinely hurts your paper.

5. After Submissions Were Received

HotCRP had 408 records of submissions, but many were just from novice authors experimenting with HotCRP to see how it works; some authors re-uploaded their paper multiple times, creating duplicate submissions; and some authors registered the paper days earlier, but withdrew or never completed their submission.

There were just over 350 complete submissions. We inspected each one of them! Not surprisingly, we found over a dozen submissions that did not follow the rules: many did not anonymize and even had the authors' full names on the title page; some had bad fonts or bad PDFs; some had odd background colors or a watermark; some violated the formatting guidelines (fonts, sizes, margins, or the number of pages). We contacted each of those papers' authors and gave them 24 hours to fix their submission or we might have been forced to withdraw their papers. Most authors complied quickly. A couple of papers were withdrawn. We were left with 348 submissions that were ready to be reviewed.

6. Conflict Management and Integrity

A conference's reputation and prestige depend heavily on at least two important factors. First, the number, length, and quality of the reviews received. This is especially important for rejected papers. Getting a paper rejected is always disappointing; a good conference will provide reviews that would show authors that reviewers understood their paper, read it carefully, and provided constructive criticism to help the authors improve the work for a future resubmission.

Second, the perceived fairness of the process. In a conference like ATC, where over 80% of all papers are rejected, it is vital that the authors of all those rejected papers would feel that the process was fair and unbiased: that the decision on papers which got accepted or rejected was based on merit alone, and not on who reviewed, who was on the PC, or who are the conference organizers.

In 2019, there were stories circulating about possible poor practices in certain systems conferences. In one case this led to a tragedy. Organizations such as ACM and IEEE are still investigating various reports. And we have also heard from other senior members of our community, directly and indirectly, about possible conflicts and violations of anonymity rules.

We expected and followed three common conflict and anonymity rules:

1. PC chairs are prohibited from submitting papers to their conference. When recruiting our Submission Co-Chairs, we told them they would also not be allowed to submit papers to ATC'20. That is because all four of us had "superuser" privileges on HotCRP, allowing us to see any review.
2. Authors had to mark every PC conflict (and select a reason) for their paper. HotCRP is excellent at ensuring that conflicted PC members had no direct or indirect knowledge about papers those PC members are conflicted with: they can't see reviews, who reviewed, or scores; and they would be excluded from all online and PC discussions regarding conflicted papers.
3. Both of us Co-Chairs had papers marked as conflicts (often from our collaborators or ex-students). That's unavoidable. Thankfully there were no papers in which we were both conflicted. Therefore, we decided that each one of us PC Co-Chairs would alone handle all matters (e.g., PC discussions and decisions) relating to papers we were conflicted on; this included getting us PC Co-Chairs (or Submission Co-Chairs) excluded from parts of the live PC meeting discussions, as well as the selection of paper awards.

Nevertheless, we decided to "take it up a notch":

First, we added a new statement to the CFP that read as follows:

> Authors and others are prohibited from directly or indirectly communicating with any ATC '20 PC/ERC member about any potentially submitted paper. All requests should be made exclusively to atc20chairs@usenix.org. Violations of these guidelines may seek remedies as stipulated in the USENIX Conference Submissions Policy.

This was to ensure that PC members and authors should never communicate directly, regardless of whether a paper was accepted or rejected, even after the conference is long passed.

Second, our Submission Co-Chairs leveraged past scripts and data-sets that scour the Web, DLBP paper databases, and other resources. Those scripts attempt to identify additional or incorrectly marked conflicts for every paper. The scripts, for example, attempt to find if person A collaborated with person B on a recently published paper, person A submitted to USENIX ATC '20, person B was a PC member, but person B was not marked as a conflict for person A's submission. The scripts also looked at possible institutional conflicts, and more.

These scripts were far from perfect. They produced many false positives due to very similar names as well as their inability to perfectly match people's names with abbreviated initials. Our Submission Co-Chairs manually poured over thousands(!) of flagged potential conflicts from these scripts and together we narrowed them down to about 242 possible conflicts across 92 papers. We wrote separate scripts that emailed all authors and asked them to verify whether these possible conflicts were valid or not. Over 50% of all responses indicated that indeed these were valid conflicts for those papers, and so we marked additional conflicts.

We were now ready in earnest to assign papers to reviewers and begin the first round of reviews.

Phew.

7. Next Stages of Running a Conference

At this stage of any conference, about half of the Co-Chairs' work is done: forming a PC, posting a CFP, getting and sanitizing papers, and assigning reviews. We could now sit back and relax for a while...

... or so we thought.

COVID-19 happened. In other words, the Borg was unleashed on us before anyone was ready.

For many of us, the amount of work in our day-jobs increased considerably. Food and supply shortages. Dire predictions every day. No one knows what happens next. Many of us who are parents were having to juggle twice the workload and now having to provide full-time childcare, with schools and childcare facilities closed. People losing their jobs in droves. The economy tanking. The stress levels everyone was going through were through the roof.

We began to receive reports from PC members who were unable to meet the already tight review deadlines. Some reviewers reported stress, work and family problems, and even health problems related to COVID-19. Not prying, we tried our best to work with everyone's abilities and schedules. In some cases, we had to reassign papers and reviews to ensure we kept the expected review quality of ATC. This was a non-stop daily effort from mid-February to mid-April of 2020.

8. The PC Meeting

An in-person PC meeting is important. No amount of written reviews and numeric scores can replace the human experience. When people discuss a paper face to face, everyone can better gauge exactly how every reviewer feels about a paper. Seeing people's faces, hearing the inflection in their voices, and noting their body language are all vital cues that help us communicate better with one another. This is especially important because the PC meeting is often reserved for the most challenging borderline papers whose fate depends on informed discussions.

We had grand plans to hold a day-long physical PC meeting at Georgia Tech. The plan was to review no more than 70 papers during the PC meeting.

We started to get many emails from PC members who were concerned about traveling due to their own health, or their employer restricted or prohibited all travel. It was clear that holding a physical conference was not possible. So we canceled it and informed all PC members that it would be a virtual PC.

We had no idea how we would run a virtual PC yet. But we realized that we would not be able to hold a day-long virtual PC, nor would we have time to review ~70 papers virtually.

Based on past experience, we had already decided to pre-accept and pre-reject some papers during the R2 discussion period (and recall we also sent out early rejections for papers that were rejected in R1). The thinking was that if the PC can reach accept/reject decisions on some papers during the R2 online discussion period, and ahead of the PC meeting, then there would be fewer papers that had to be discussed at the actual PC. Past experience suggested that such papers' decisions rarely change during a PC meeting, so it was just going to waste time if we discussed them. Therefore, we assigned a discussion leader to each paper and asked them to see if a consensus could be reached during the R2 discussion period: pre-accept, pre-reject, or "need to be discussed at the PC." We gave PC members wide latitude to opine on those pre-accepted and pre-rejected papers. If any reviewer felt the decision was incorrect, then that paper would be discussed at the PC instead.

In normal times, it's just too easy to "punt" on making difficult decisions and merely push the paper to be discussed at the PC meeting. But now, having been forced to run a shorter, virtual PC, we could not afford to leave too many papers for the PC meeting itself. So we pushed our PC to make the difficult decisions early on, try to reach consensus, and pre-accept or pre-reject as many papers as possible. In order to provide for some cross-paper "calibration," we configured HotCRP so as to make all reviews and discussions visible to all non-conflicted PC members. This was difficult for everyone, especially during a worldwide pandemic. But our PC did admirably well. We were left with 38 papers to discuss at the PC meeting.

8.1. The Virtual PC (vPC)

Running a virtual PC (vPC) was a new thing. Few have ever attempted it. We could write ten more pages of "Chair Notes" about our experience. Wait, we did! You can find our full-length report on how we planned and ran our vPC here: https://www3.cs.stonybrook.edu/~ezk/vPC.html. So we'll only give you the highlights below:

1. We (two Co-Chairs and two Submission Chairs) evaluated multiple solutions and in the end, we chose Zoom as the most suitable one.
2. We shortened the PC meeting to just five hours. But we underestimated how long it would take. Indeed, we ran about 2 hours longer.
3. We polled our PC members for their schedules and current time-zones. And we clustered the papers to discuss into groups that best optimize for PC members' availability.
4. In a physical PC meeting, PC members who are conflicted are asked to leave the room, then they're called back in. With a vPC, we used Zoom's "Waiting Room" feature: it neatly allowed us to virtually move conflicted PC members from the actual PC to the waiting room, where their video and audio are disabled, they cannot hear or see any of the discussions, until they are let back in.

Given the unprecedented circumstances we faced, we believe the vPC went very well. And we strongly believe that USENIX ATC '20 maintains the high standards of quality despite all that everyone had to go through.

8.2. The Stats

And now, for the customary statistics from the paper review process.

Finalizing the program decisions for USENIX ATC '20 took 1,379 reviews, 4,193 discussion comments posted during the two online discussion rounds (post R1 and post rebuttals). The PC wrote a total of over 1.15 million words. Out of the initial 348 submissions, 150 papers (43%) were advanced to the second round, the remaining 198 papers received an early (R1) rejection notification. During the post-rebuttal online discussion period, the PC decided to pre-accept 42 highly-ranked papers and to pre-reject 70 more papers. The remaining 38 papers were further discussed during the vPC meeting. 23 of these papers were accepted and 15 were rejected. The final acceptance rate for ATC '20 is 18.7% (65 out of 348).

Among the accepted papers, 61 are regular full-length papers, 3 are short papers which were submitted as short, one paper will appear at the conference as a short paper but was originally submitted as a full-length 11-page paper, and another paper submitted as long was accepted as a short paper but given 9 pages. All accepted papers were initially conditionally accepted and assigned 1 or 2 shepherds, who ensured that the final version of the paper addressed the reviewers' feedback.

9. The Conference

With the paper selection process complete, we were almost done—just to organize the papers in sessions that fit the more-or-less standard schedule template that USENIX was going to provide…

… or so we thought.

Shortly after the vPC concluded and all decisions were communicated, unsurprisingly, it was determined that the actual conference will also have to be held as a virtual event. So, we were back in uncharted territory, trying to make the best decisions on how to organize and run a multi-day, multi-track USENIX ATC '20 conference—and for the first time fully online.

We spoke with Co-Chairs of other major systems conferences that switched to virtual mode—ASPLOS '20, Eurosys '20, and ISCA '20. We polled some of our PC members for their feedback from virtual conferences they had attended. We read blogs and recommendations on different virtual conference tools and experiences. We consulted on a weekly basis with the USENIX staff and the Co-Chairs of the workshops affiliated with ATC (HotStorage and HotCloud). And we mapped out the time zones of the authors of the accepted papers, and much more.

In the end, we arrived at a program that retained the original conference dates, organized in 12 technical papers sessions presented in two parallel tracks, and two plenary keynote addresses. With a goal of creating a balance among giving to authors an opportunity to present their work in front of a live audience, providing conference attendees with high-quality technical content, and shielding everyone from "Zoom fatigue," we opted for a schedule that combines longer, asynchronously-delivered, pre-recorded presentations, and live sessions with shorter video presentations and Q&A.

The two keynotes are scheduled at the end of the first two days of the conference, with ample time for discussions and Q&A. The first keynote is by Professor Ethan Miller from UC Santa Cruz on "The Future of the Past: Challenges in Archival Storage" and discusses, for example, future storage technologies such as DNA and Glass. The second keynote is by Professor Margo Seltzer from the University of British Columbia, titled "The Fine Line between Bold and Fringe Lunatic." Margo's talk hopefully sets a new tradition at USENIX ATC where the previous year's Lifetime Achievement Award Winner delivers one of the keynote addresses in the following year.

We look forward to three days of technical papers, keynote presentations, and discussions.

10. In Closing

Despite the tremendous amount of (unanticipated) work, we are thrilled to have had the honor of chairing USENIX ATC. We are thankful to everyone who helped and contributed along the way: to the authors who submitted their high-quality work to ATC, to the dedicated program committee and external reviewers who evaluated hundreds of submissions and provided constructive feedback to the authors, to the PC Leaders, Submission Co-Chairs and to the USENIX staff who provided invaluable advice and support.

We are excited to welcome everyone at the first-ever virtual USENIX Annual Technical Conference, and we hope you will enjoy it.

USENIX ATC '20 Program Co-Chairs
Erez Zadok, *Stony Brook University*
Ada Gavrilovska, *Georgia Institute of Technology*

Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface

Jungsik Choi
Sungkyunkwan University

Jaewan Hong
KAIST

Youngjin Kwon
KAIST

Hwansoo Han
Sungkyunkwan University

Abstract

Fast non-volatile memory (NVM) technology changes the landscape of file systems. A series of research efforts to overcome the traditional file system designs that limit NVM performance. This research has proposed NVM-optimized file systems to leverage the favorable features of byte-addressability, low-latency, and high scalability. The work tailors the file system stack to reduce the software overhead in using fast NVM. As a further step, NVM IO systems use the memory-mapped interface to fully capture the performance of NVM. However, the memory-mapped interface makes it difficult to manage the consistency semantics of NVM, as application developers need to consider the low-level details. In this work, we propose Libnvmmio, an extended user-level memory-mapped IO, which provides failure-atomicity and frees developers from the crash-consistency headaches. Libnvmmio reconstructs a common data IO path with memory-mapped IO, providing better performance and scalability than the state-of-the-art NVM file systems. On a number of microbenchmarks, Libnvmmio gains up to 2.2× better throughput and 13× better scalability than file accesses via system calls to underlying file systems. For SQLite, Libnvmmio improves the performance of Mobibench and TPC-C by up to 93% and 27%, respectively. For MongoDB, it gains up to 42% throughput increase on write-intensive YCSB workloads.

1 Introduction

The recent surge of non-volatile main memory (NVM) technology such as PCM [32, 55], STT-MRAM [4, 30], NVDIMMs [45], and 3D Xpoint memory [21] allows applications to access persistent data via CPU `load`/`store` instructions directly. With the benefits of competitive performance, low power consumption, and high scalability, they are expected to complement or even replace DRAM in future systems [30, 33].

To leverage the performance and persistent features, researchers have proposed NVM-optimized file systems [8, 12, 13, 24, 28, 46, 65, 67, 68]. The most important challenge addressed in the series of work is to revise the inefficient behavior of the software IO stack, which presents a dominating overhead in fast NVM [2, 3, 9, 22, 26, 48, 69]. To reduce the overhead, state-of-the-art NVM-aware file systems discard the traditional block layer and the page cache layer in the IO path. Despite these optimizations, file accesses through the OS kernel's file system still incur significant overhead. For example, `read` and `write` system calls are still expensive ways to leverage the low latency of NVM, due to frequent user/kernel mode switches, data copies, and complicated VFS layers [7, 9, 24, 25, 27, 57, 62].

A promising approach to further reduces IO overhead of NVM file systems is to use memory-mapped IO [9, 35, 58, 60, 67, 68]. The memory-mapped IO naturally fits the characteristics of NVM. Applications can map files to their virtual address space and access files directly with `load`/`store` instructions without kernel interventions. Memory-mapped IO also minimizes the CPU overhead of file system operations by eliminating file operations such as indexing to locate data blocks and checking permissions [65]. With these benefits, the `mmap` would be a critical interface for file IO in future NVM systems.

While memory-mapped IO exposes the raw performance of NVM to applications, a lot of responsibility is laid on applications as well. One thing to keep in mind for application programmers is that memory-mapped IO does not guarantee atomic-durability. If a system failure occurs during memory-mapped IO, the file contents may be corrupted and inconsistent in the application context. In return for fast performance, developers should build application-specific crash-safe mechanisms. Cache lines should be flushed to ensure durability and memory barriers should be enforced to provide a correct persistent ordering for NVM updates. This mechanism often induces a serious software overhead, and makes it notoriously difficult to write accurate and efficient crash-proof code for NVM systems [38, 50–52, 71]. For an instance, applying cache flush and memory barrier instructions correctly in the right locations is challenging; excessive use causes performance degradation, but omitting them in required locations leads to data corruption [39, 70]. This is the major obstacle blocking the adoption of memory-mapped IO to fully exploit the advantages of NVM.

We propose *Libnvmmio*, a user library that provides failure-atomic memory-mapped IO with `msync`. We add atomicity and ordering features to the existing `msync` at user-level. By separating failure-atomicity concerns from memory-mapped IO applications, Libnvmmio allows developers to focus on the main logic of programs. To make the `msync` failure-atomic, Libnvmmio uses user-level logging techniques. Our library stages written data to per-block, persistent logs and applies the updates to memory-mapped files in a failure-atomic manner on `msync`.

Implementing `msync` at user-level has many advantages. First, the user-level `msync` minimizes system call overhead. Existing `msync` imposes system call overhead, which takes locks and excessively serializes threads in a multi-threaded application. Second, it reduces write amplification. Kernel-level `msync` flushes rather large ranges whose size are multiples of the system page size (4KB, 2MB, or 1GB). Whereas, user-level `msync` can track dirty data at a cacheline granularity and flush them at cacheline level. Third, it avoids TLB-shootdown overhead. When applications invoke `msync` on NVM file systems, operating systems track down updated pages by searching for dirty bits in the page table and flush corresponding cache lines of those dirty pages to NVM. After the flush, they clear the dirty bits in the page table to enable tracking new updates. This incurs TLB invalidations in other cores, as dirty bit state is just kind of information in TLB along with the virtual to physical page mapping. As Libnvmmio's `msync` maintains user-level logs for update tracking, we can totally avoid TLB-shootdown overhead. Fourth, it takes advantage of non-temporal `store` instructions which bypass CPU caches with no need of cache flushing. Kernel-level `msync` flushes the entire range, even if updates are performed with non-terminal `store` instructions. In general, there is no other way to communicate with `msync` that the non-temporal stores are used. For all of these reasons, a user-level `msync` in Libnvmmio can perform better than a kernel-level `msync`.

Existing applications that use conventional file IO interface (*e.g.*, `read`/`write`, `fsync`, *etc.*) can also benefit from memory-mapped IO using Libnvmmio. Like FLEX [66] and SplitFS [24], Libnvmmio transparently intercepts the traditional file IO requests and then perform memory-mapped IO. When applications call `fsync`, Libnvmmio carries out its failure-atomic `msync`. Libnvmmio rebuilds the common IO path with efficient mechanisms for read and write performance, but the uncommon, complex file operations such as directory namespace and protection are passed to the slow path of the existing file systems.

Libnvmmio runs on any file systems that supports memory-mapped interface on NVM such as Ext4-DAX, XFS-DAX, PMFS [13], and NOVA [68]. Libnvmmio running on NOVA performs better than NOVA by 2.5× and Ext4-DAX by 1.18× in Mobibench and TPC-C.

Libnvmmio makes the following contributions:

- Libnvmmio extends the semantics of `msync`, providing failure-atomicity.
- With experimental evidences, Libnvmmio demonstrates lower-latency and higher-throughput with scalability than the state-of-the-art NVM file systems
- Design and implementation of Libnvmmio, running on Ext4-DAX, XFS-DAX, PMFS, NOVA. Libnvmmio is publicly available at: https://github.com/chjs/libnvmmio.

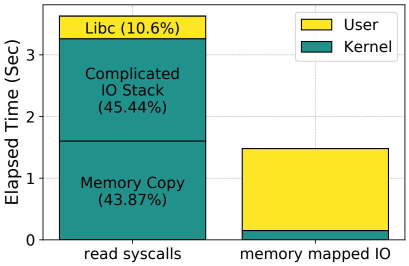


Figure 1: Read syscalls vs. memory mapped IO. Sequential read on a 16GB file. Both cases use `read` or `memcpy` to copy file data into the user buffer by 4KB.

2 Background

2.1 Need for Memory-Mapped IO

The fundamental difference between memory-mapped IO and read-write IO is the data path. The read-write interface copies the user buffer into a kernel buffer[1], searches the file system index to locate physical block address, and performs metadata operations if necessary. Whereas, the memory-mapped interface allows direct accesses to storage, skipping the index searching and copying to the kernel buffer. The simplified data path in memory-mapped IO drastically reduces the software overhead compared to the read-write interface, which significantly improves IO performance in fast non-volatile memory. To compare the performance, we run a micro-benchmark performing sequential reads on a 16 GB file. Figure 1 shows the performance difference. Memory-mapped IO shows 2.3× better performance than the `read` system call. The `read` system calls spends 43.9% out of the IO entire latency on copying user buffers to kernel buffers and 45.4% for the rest of kernel IO stack. Memory-mapped IO eliminates most of the software overhead. We observed that the total number of instructions to execute a single read

[1] Some NVM file systems such as NOVA avoid it.

is 69× less in the memory-mapped IO than the `read` system call.

2.2 Need for Atomic Updates

Modern processors guarantee only cache-sized, aligned stores (64 bit) to be atomic. The atomicity guarantee is not sufficient for general file IO which requires more complex and larger atomic updates. On writing a 4 KB or larger block, a crash may cause partially updated states, which needs significant costs to detect and recover the block. To avoid the hassle, researchers put an effort to make large updates *failure-atomic* in non-volatile memory file systems [24, 28, 67]. Existing file systems deploy a variety of techniques to implement the failure-atomicity guarantee: copy-on-write and journaling. These techniques work in different ways, and the advantages and disadvantages in terms of performance vary.

2.2.1 Copy-on-Write

When updating a block, the Copy-on-Write (CoW) (or shadow-paging) [12, 17, 42, 56, 67, 68] mechanism creates a copy of the original page and writes the new data to the copied page rather than updating the new data in place. Not only for data update but the CoW mechanism performs the out-of-place update for index. For a tree-based indexing structure, the CoW mechanism causes a change of a child node to update its parent node in an out-of-place manner, propagating all the changes of internal nodes up to the top of the tree (called *wandering tree problem*).

The CoW mechanism induce significant software overhead when used in the NVMM system. First, CoW dramatically increases write amplification. CoW usually performs writes at the page granularity, which is a typical node size of file systems indexing. Even if only a few bytes are updated, the entire page must be written. Besides, as the capacity of main memory has increased, the utilization of hugepages (*e.g.*, 2MB or 1GB) is increasing [6, 13, 14, 29, 47, 54]. This trend makes the use of the CoW technique more costly [9]. Second, the CoW technique causes TLB-shootdown overhead in memory-mapped IO. If the CoW technique is applied to memory-mapped files, the mapping of the virtual address must be changed from the original page to the copied page, necessitating TLB-shootdown whenever an update occurs. When a CoW occurs, the kernel flushes the local TLB and send flush requests to remote cores through inter-processor interrupt (IPI). The remote cores flush their TLB entries according to the information received by the IPI and report back when completed. If the remote core has interrupts disabled, the IPI may be kept pending. The initiator core expects to receive all acknowledge the process of flushing the TLBs. This process could take microseconds, causing a notable overhead [3, 61].

2.2.2 Journaling

Journaling (or logging) is a technique that is widely used in databases [43] and journaling file systems [13, 16, 22, 34, 49, 53] to ensure data-atomicity and consistency between data and metadata. It persists a copy of new or original data before updating the original file. If a system failure occurs during writing, the valid log can be used for recovery. Two logging policies are possible: undo logging and redo logging. Redo logging first writes new data to the redo log. When the new data becomes durable in the log, the data are overwritten to the original file. If a system failure occurs while updating the file, the new data in the log can be written again to the file. For read requests, applications need to check the log first because only the log may have the up-to-date data. Undo logging first copies the original data to the log. After the original data becomes persistent, undo logging updates the new data to the file in place. If a system failure occurs during the write, undo logging allows to roll back the original data using the undo log. Because the latest data are always in the file, applications can read the data directly from the file without checking the log. Therefore, undo logging is appropriate for the applications that perform read frequently (§3.4).

Logging techniques require writing data twice: once to the log and once to the original file, which may cause software overhead. However, redo logging allows updating the original file out of the critical path of execution. Because the log has the persistent data, redo logging can postpone updating the file in the background (§3.3). Besides, logging technique is convenient to implement the *differential logging* [1,15,23,36]. Unlike page-based logging, which logs an entire page, the differential logging only logs differential data at the byte-granularity. Differential logging can significantly reduce write amplification especially when it is used for byte-granularity storage devices such as NVM [27].

2.3 Atomic Update for Memory-Mapped IO

While the direct access of memory-mapped IO is essential for reducing the software overhead in NVM file system, it pushes the burden of data atomicity to the application. The POSIX `msync` primitives provides durability and consistency between data and metadata but not atomicity. To support atomicity of large updates, application developers must implement their own reliability mechanism. However, implementing the in-house mechanism is tedious and notoriously buggy [50].

Researchers have proposed adding the atomicity guarantee to the `msync` interface in traditional storage [50] and NVM [67]. To provide atomicity to memory-mapped files, they take journaling-like approaches; dirty pages are staged first and copied to the original file. Providing atomicity at the kernel-level has a fundamental limit which impacts good performance. For example, NOVA [67] creates a replica page on a page fault and maps the replica page on the faulting virtual address. On msync, kernel copies the replica page to the original page atomically. The minimum unit of copying is a page size (4 KB or 2 MB), which causes write amplification for small IO requests.

3 Libnvmmio

The purpose of Libnvmmio is eliminating software overhead, while providing low-latency, scalable file IO with ensured data-atomicity. Libnvmmio is linked with applications as a library, providing the efficient IO path by using the mmap interface. In particular, Libnvmmio has following design goals and implementation strategies.

Low-latency IO. Reducing software overhead is crucial to take advantage of low latency NVM. Since Libnvmmio aims to make the common IO path efficient for low-latency IO, it avoids using the complicated kernel IO path including the slow journaling for common cases.

Efficient logging for data atomicity. Libnvmmio transparently intercepts file APIs and provides atomicity for data operations by using logging. As sustaining low-latency file IO is essential, Libnvmmio endeavors to minimize write amplification and software overhead for data logging.

High-throughput, scalable IO with high concurrency. To sustain high throughput across different IO sizes, Libnvmmio uses varying sizes of log entries depending on IO sizes. To this end, Libnvmmio deploys a flexible data structure for indexing the log entries and handles various *log entry* sizes. Additionally, Libnvmmio aims to achieve high concurrency through fine-grained logging and scalable indexing structure.

Data-centric, per-block based organization. Libnvmmio constructs most of its data structures and metadata as data-centric. For example, Libnvmmio builds per-block logs and metadata rather than per-thread or per-transaction based logs. Data-centric design allows a single instance of a data structure and metadata for a corresponding data block. The singleton design makes it easy to coordinate shared accesses with locks. As multiple threads access the same large file concurrently in recent applications, they require more fine-grained locks than entire file locks [40]. With fine-grained locks at block level, Libnvmmio achieves scalability for data-centric logging. Per-inode logging improves scalability, when multiple accesses are performed on different files [67,68]. However, it provides a limited degree of scalability for multiple accesses to the same file.

Transparent to underlying file systems. On top of existing NVM file systems, Libnvmmio improves the performance for common data IO, keeping POSIX interfaces unchanged. For complex, uncommon IO operations, Libnvmmio leverages rich, well-tested features of existing file systems. Without breaking POSIX semantics, Libnvmmio offers extended POSIX APIs to applications for additional features. For example, POSIX semantics does not guarantee atomicity of mmap. While atomicity is useful, not all files need atomic update guarantees — it is unnecessary for temporal files. Libnvmmio extends open API to let applications indicate atomicity guarantee in a per-file basis. To communicate with the kernel, Libnvmmio translates the extended APIs to the conventional APIs with additional flags. With such a user-level extension design, Libnvmmio runs on any NVM file systems that support DAX-mmap, while enjoying file-system specific features such as fast snapshot and efficient block allocation.

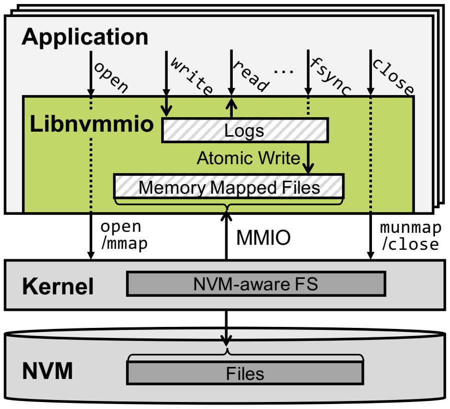


Figure 2: Libnvmmio Overview

3.1 Overall Architecture

Libnvmmio runs in the address space of a target application as a library and interacts with underlying file systems. Libnvmmio intercepts IO requests and turns them into internal operations. For each IO request, Libnvmmio distinguishes data and metadata operations. For all data requests, Libnvmmio services them in the user-level library, bypassing the slow kernel code. Whereas, for complex metadata and directory operations, Libnvmmio lets the operations be processed by the kernel. This design is based on the observation that data updates are the common, performance-critical operations. On the other hand, the metadata and directory operations are relatively uncommon and include complex implementation to support POSIX semantics. Handling them differently, the architecture of Libnvmmio follows the design principle of making the normal case fast [31] with a simple, fast user-level implementation.

Figure 2 shows the overall architecture of Libnvmmio. When an application opens a file, Libnvmmio interposes the open call with a user-level open API. Within the open API, it maps the whole content of the file onto the user memory space and initializes *per-file metadata* (§3.5). The metadata Libnvmmio initializes includes inode number, logging policy, epoch number, *etc*. After the initialization, it returns the file descriptor to the application.

Memory-mapped IO. To directly access the NVM, Libnvmmio maps the file via `mmap` system call. Libnvmmio intercepts and replaces `read` calls with `memcpy`, and `write` calls with a non-temporal version of `memcpy` that uses the `movnt` instruction. There are two reasons why the memory-mapped IO allows faster NVM access than the traditional kernel-served read and write method. First, when persisting and obtaining data, the simple, the fast code path in Libnvmmio replaces the complex, slow kernel IO path [24, 28]. Second, `read` and `write` system calls involve indexing operations to locate physical blocks, which causes a non-trivial software overhead for fast NVM accesses. Whereas, in memory-mapped IO, the kernel searches the complex index when it maps the file blocks to the user address space on page faults. After the mapping is established, Libnvmmio can access the file data simply with offset in the memory-mapped address, eliminating the indexing operations in the steady state. Besides, finding file blocks through virtual addresses is offloaded to the MMU (*e.g.*, page table walkers, TLBs). Therefore, it reduces a sizable amount of the CPU overhead caused by file indexing [65].

Atomicity and durability with user-level logging. On `SYNC`[2] calls, Libnvmmio flushes the cache data and stores the data to NVM atomically via the logging mechanism. All write data are firstly persisted to the user-level log and later they are copied (called *checkpoint*) to the memory-mapped file. Data from both `write` and `memcpy` interfaces goes down the same path.

Providing atomicity via the user-level logging has several advantages over the kernel-level design. Using the user-level IO information, Libnvmmio can leverage the byte-addressability of NVM to log data in the fine-grained unit. On the other hand, in the kernel-level approach, the logging unit should be a page size, as `msync` relies on the page dirty bit to log the memory-mapped data, causing write amplification in case of small writes (*i.e.*, less than a page size). After `msync` is done, kernel must clear the dirty bit in the page table followed by TLB shootdown. However, user-level design uses own data structure to track dirty data without relying on the page dirty mechanism, saving unnecessary TLB shootdowns.

Application transparency. For applications using `read` and `write`, Libnvmmio can transparently replace them with the memory mapped IO operations. For applications using `mmap`, Libnvmmio can redirect the memory operations to NVM memory-mapped IO operations without effort.

Providing atomic-durability on top of the `mmap` interface makes the case challenging, as Libnvmmio cannot distinguish the `memcpy` operations that requires atomic-durability from the ones that do not require.

Guaranteeing atomicity to all IO operations is prohibitively expensive. Some IO requests do not need atomicity such as logging internal traces or errors. To address the problem, Libnvmmio exposes two version of `memcpy`: POSIX version and Libnvmmio version. Libnvmmio versions are prefixed with `nv` (*e.g.*, `nvmmap`, `nvmemcpy`, `nvmunmap`, *etc.*) and provide atomic-durability. Libnvmmio avoids intrusive modifications of existing applications in order to use the Libnvmmio APIs. Instead, we instrument the application binary with an in-house tool, which lists the files the application accesses and asks developers which files need atomic-updates. With the list of files requiring atomic-durability, we patches the binary to use Libnvmmio APIs. In most cases, applications use `read`, `write`, or `memcpy` APIs, which are easy to patch for the application binary. However, in case of manipulating files with pointers, we need source-level modifications (*e.g.*, 182 lines in the MongoDB MMAPv1 engine).

3.2 Scalable Logging

Applications such as in-memory database and key-value stores, that benefit from Libnvmmio, require high concurrency level to sustain high throughput. Libnvmmio responds to the high concurrency requirement with scalable logging that is based on per-block data logging and indexing.

3.2.1 Scalable per-block logging

Finding proper logging granularity is necessary to achieve high concurrency. Application-centric techniques such as per-thread and per-transaction logging are widely adopted in databases, providing high concurrency. However, these techniques rely on the strong assumption that data is only visible and applicable to the current thread or transaction; *e.g.*, data in logs need not to be shared among threads or transactions, which is guaranteed by isolation property. Logging without needing to consider shared data allows for high scalability. However, the assumptions do not hold in general IO cases; sharing IO data among threads is a common use case. Moreover, the transaction boundary is not visible to the current design of Libnvmmio.

Instead, Libnvmmio performs data-centric logging. It divides the file space into multiple file blocks (4 KB∼2 MB) and creates a log entry for each file block. Log entries in Libnvmmio are visible to all threads. The fine-grained, per-block logging allows a flexible way to share data among threads. When an update is made to a mapped file, Libnvmmio creates a log entry indexed by the offset, where the update occurred in the memory-mapped file. If other threads read the updated offset, it serves data from the log entry instead of the original

[2] This term means both `fsync` and `msync`.

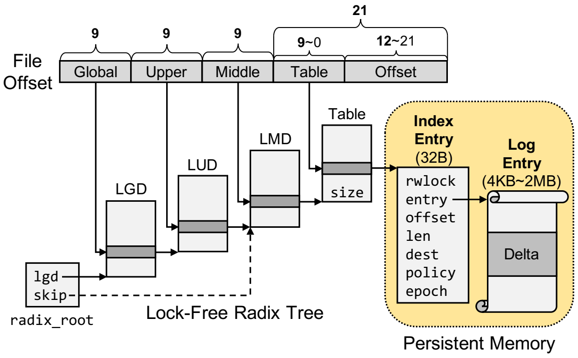


Figure 3: Indexing structure of Libnvmmio.

mapped file. When another update comes to the same file offset, it overwrites the update in the existing log entry. For shared data reads, per-block logging provides better performance than per-thread logging, as per-thread logging needs to search all the logs of all threads to gather all the updates made to the same file blocks. In addition to per-block logging, Libnvmmio takes advantage of the byte-addressable characteristics of NVM and reduces write amplification by performing differential logging for a partial update, where the update size is smaller than log block size.

3.2.2 Scalable log indexing

Along with data logging, indexing design is also critical to achieve high concurrency. Libnvmmio uses a file offset as an index key to a log block. To index many log blocks, Libnvmmio uses multi-level indexing to reduce space overhead. Similar to the page table, it uses radix trees for indexing. Fixed-depth trees allow lock-free mechanisms, which provide better concurrency than balanced trees such as red-black trees. As balanced trees require coarse-grained locks to protect the entire trees for tree re-balancing, their algorithms severely hurt concurrency [10, 11].

Figure 3 shows the index design of Libnvmmio. Each internal node is an array of buckets pointing to the next level internal nodes. Each set of 9 bits from file offset is used to locate a bucket in a corresponding internal node. Each leaf node points to an *index entry*, where entry field points to *log entry*. The index entry also contains other metadata for the given file offset. Libnvmmio supports variable-size log entries for large IO requests. Log entries range from 4KB to 2MB, doubling the size. To index 4KB log entries, it uses 9 bits for Table and 12 bits for Offset. For 2 MB log entry, it uses 21 bits for Offset without using Table.

In an index entry, offset and len are used for updated data offset within a log entry and update size, respectively. If update size in len is smaller than the log entry size, it means the log entry contains partial updates (Delta). The log entry can hold a single delta chunk indicated by offset and len. If another delta chunk needs to be added in the same log entry, the two chunks are merged. The virtual address of the memory mapped file specified in dest is the location where the log will be checkpointed. The logging policy for the corresponding data is specified in policy, which decides whether Libnvmmio uses undo log or redo log (§3.4). To determine if the log entry should be checkpointed, the number in epoch is used (§3.3).

The radix tree has a fixed depth to implement a lock-free mechanism. The four-level radix tree can support 256 TiB file size, but it can cause unnecessary search overhead for small files. Libnvmmio uses a skip pointer to implement a lock-free radix tree while also reducing the search overhead. As shown in Figure 3, the radix_root has a skip field. If the file size is small, Libnvmmio uses the field to skip unnecessary parent nodes. When the file size changes, Libnvmmio can adjust the skip pointer.

To achieve fast indexing, Libnvmmio manages the internal nodes of the radix tree in DRAM and does not persist them to NVM. It persists only the index entries and the log entries. Libnvmmio does not need to build the entire radix tree for recovery. On a crash, it simply scans the persisted index and log entries, which are committed but not checkpointed yet. It can copy the log entries to the original file by referring the dest attribute in the corresponding index entries and the per-file metadata. To achieve high concurrency, Libnvmmio does not use any coarse-grained locks to update internal nodes of the radix tree. Instead, it updates each bucket of internal nodes with an atomic operation. Only when it needs to update index entry, it holds the per-entry, reader-writer lock.

3.3 Epoch-based Background Checkpointing

Log entries are committed on SYNC[3]. The committed log entries must be checkpointed to the corresponding memory-mapped file and cleaned. To make the checkpoint operations out of the performance critical path, Libnvmmio checkpoints the log entries in the background. It periodically wakes up checkpointing threads for copying and cleaning log entries[4]. While checkpointing, the background threads do not need to obtain a coarse-grained tree lock. This minimizes disruption on on-going read/write operations. The background threads holds a per-entry writer lock to serialize checkpoint operations and read/write requests on the log entry.

Libnvmmio uses per-block logging. When an application calls SYNC, it must convert many of the corresponding per-block logs to committed status. This increases the commit overhead significantly. To avoid such overhead, Libnvmmio performs committing and checkpointing based on the epoch, which increases monotonically. Libnvmmio maintains two

[3] This term means both fsync and msync.

[4] Through sensitivity studies, we configured Libnvmmio wakes up the threads every 100 microsecond.

types of epoch numbers; each index entry has an epoch number for its update log and per-file metadata carries the current global epoch number. When allocating an index entry, it assigns the current global epoch number for file to the epoch number for the index entry. Libnvmmio increases the current global epoch number, when applications issue SYNC calls to the file. The epoch numbers are used to distinguish committed (but yet to be checkpointed) log entries from the uncommitted ones. If a log entry has a smaller epoch number than the current global epoch number, it indicates that the log entry is committed. If the epoch number of a log entry is the same as the global epoch number, the log entry is not yet committed. Libnvmmio checkpoints only committed log entries in the background threads. After being checkpointed, log entries are cleaned and reused later.

The epoch-based approach allows fast commit of log entries, as Libnvmmio does not need to traverse the radix tree and mark log entries as committed. Instead, it simply increases the current global epoch number in the per-file metadata, which reduces SYNC latency greatly. Commit operations are performed synchronously and atomically, when the application calls SYNC. Meanwhile, checkpoint operations are done asynchronously by background threads. Consequently, there are committed logs and uncommitted logs mixed in the radix tree. When applications request writes, the corresponding log entries are overwritten for uncommitted ones. Meanwhile, Libnvmmio synchronously checkpoints the committed logs first for committed ones. After completing the checkpointing, it allocates a new uncommitted log and processes write requests.

3.4 Hybrid Logging

Libnvmmio uses a hybrid logging technique to optimize IO latency and throughput. As pointed out in §2.2.2, undo logging performs better when accesses are mostly reads, whereas redo logging is better when accesses are mostly writes. To achieve the best performance of both logging policies, Libnvmmio transparently monitors the access patterns of each file and applies different logging policies depending on current read and write intensity.

Libnvmmio maintains counters to record read and write operations for a file (§3.5). When SYNC is called, Libnvmmio checks the counters to determine whether which type of logging would be better for the next epoch. If the logging policy changes, Libnvmmio carries out both committing and checkpointing synchronously. SYNC is a clean transition point for changing the logging policy, as current log data are checkpointed and cleaned. This allows Libnvmmio to avoid complex cases where it otherwise has to maintain two log policies at the same time. The per-file, hybrid logging enables the fine-grained logging policy, allowing Libnvmmio to adopt the individually best logging mechanism for each file. By default, Libnvmmio uses undo logging. It switches to redo logging, when the ratio of write operations becomes higher than or equal to 40%. The policy for the new epoch is determined by the write ratio in the previous epoch. The threshold ratio is obtained from the sensitivity analysis in §4.2.1.

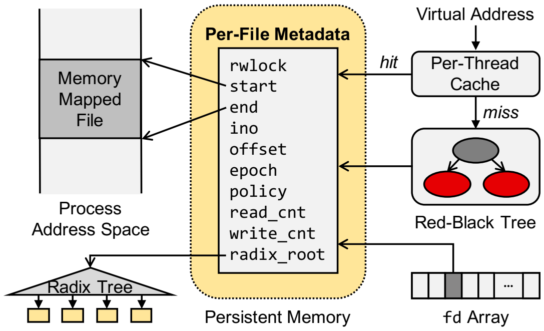


Figure 4: Per-File Metadata

3.5 Per-File Metadata

Libnvmmio maintains two types of metadata in persistent memory; the index entry is the metadata for each log entry, and the per-file metadata shown in Figure 4 is the metadata for each file. Libnvmmio stores both metadata as well as log entries in NVM, which enables Libnvmmio to recover its data in case of system failures.

When Libnvmmio accesses a file, it first gets the per-file metadata of the file and the index entry corresponding to the file offset. If applications access a file with nvmemcpy interface, it needs to find the per-file metadata by using access address of the nvmemcpy. The approach Libnvmmio takes for this purpose is to employ a red-black tree and perform range searches with virtual addresses. To speed up the search process, Libnvmmio caches recently used per-file metadata in the per-thread cache. Meanwhile, Libnvmmio can quickly obtain the per-file metadata through the file descriptor, if applications access files with read/write interface.

The per-file metadata consists of ten fields. The rwlock is a reader-writer lock. During SYNC process, this lock prevents other threads from accessing the file. The start and end fields store the location of the virtual address to which the file is mapped. The ino and offset fields record which part of a file is mapped. The epoch field stores the current global epoch number for the file. The policy field stores the current logging policy for the file. The read_cnt and write_cnt are counters of read and write operations during the current epoch, respectively. The radix_root field stores the root node of the radix tree indexing for index entries and log entries.

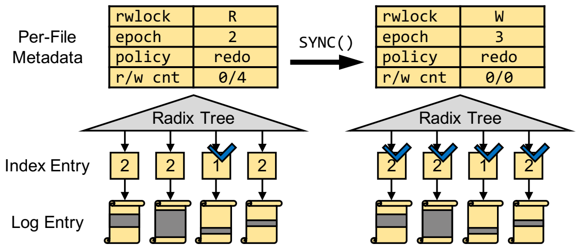


Figure 5: Epoch-based committing

3.6 Putting all together: `write` and `SYNC`

Figure 5 shows the steps of the epoch-based checkpointing in Libnvmmio. The numbers in the index entries indicate per-entry epoch numbers, and the check marks indicate their log entries are committed. A simplified version of per-file metadata is shown in tables.

Write. ① The thread holds the reader lock in the per-file metadata of the file and increases the write counter with atomic operations. Holding the reader lock in per-file metadata allows multiple threads to access the file concurrently. ② The thread traverses the in-memory radix tree to locate the corresponding index entry and holds the writer lock for the index entry. ③ Depending on the current logging policy in the per-file metadata, Libnvmmio creates an undo or redo log entry. ④ The thread writes data to the log entry with the non-temporal `store` instruction, and Libnvmmio updates the index entry of the log entry. ⑤ Libnvmmio calls `sfence` indicating logging is done and unlocks the index entry and per-file metadata, and returns to the application.

SYNC. ① Libnvmmio holds the writer lock in the per-file metadata and increases the global epoch counter by one. Holding the writer lock of the per-file metadata prevents other threads from accessing the file. ② Libnvmmio calculates the write ratio from the write and read counters. In the example in Figure 5, Libnvmmio continues to use redo logging for the next epoch, as the access pattern is write-intensive (4 writes out of 4 accesses). After determining the logging policy, Libnvmmio initializes the counters. When logging policy is unchanged, Libnvmmio lets checkpointing threads commit log entries in the background. If Libnvmmio decides to change logging policy, it synchronously checkpoints all committed log entries before the new epoch begins. ③ Finally, Libnvmmio unlocks the per-file metadata and returns to the application.

3.7 Crash Consistency and Recovery

Libnvmmio preserves write ordering of a sequence of write requests. For each write, Libnvmmio writes data to the log and flushes the CPU cache. The order-preserving write provides the prefix semantics [63], guaranteeing every thread to see a consistent version of data updates. Along with the consistency of data, Libnvmmio guarantees consistency between metadata and data. Libnvmmio maintains two persistent metadata: per-file metadata and index entries. Libnvmmio strictly orders between the sequence of [data update, index entry update] and `SYNC` call.

In the recovery phase, Libnvmmio checks whether the index entries are committed, while scanning the index entries. If Libnvmmio finds a committed log, whose epoch number is smaller than the global epoch number, it finds the per-file metadata from the index entry's `dest` attribute. Then, it redoes or undoes according to the logging policy. Libnvmmio can efficiently parallelize this recovery task by using multithreading.

NVMM	Rand Read	Rand Write	Seq Read	Seq Write
NVDIMM-N	35.84	20.61	92.42	20.65
Optane DC	3.588	1.026	13.64	4.30

Table 1: NVMM Characteristics (GB/s)

4 Evaluation

We implemented Libnvmmio from scratch. Our prototype of Libnvmmio has a total 3,452 LOC[5] in C code. To persist data to NVM, Libnvmmio employs the PMDK library [20].

4.1 Experimental setup

To evaluate Libnvmmio on different types of NVM, we used NVDIMM-N [45] and Intel Optane DC Persistent Memory Module [19]. The system with 32GB NVDIMM-N has 20 cores and 32GB DRAM. Another system with 256GB Optane has 16 cores and 64GB DRAM. In the Optane server, we used two 128GB Optanes configured in *interleaved App Direct* mode. Table 1 shows the results of measuring the performance of each memory using Intel Memory Latency Checker (MLC) [18].

In our experiment, Libnvmmio used NOVA [68] running on Linux kernel 5.1 as its underlying file system. To compare Libnvmmio with various file systems, we experimented with four file systems: Two of these, Ext4-DAX and PMFS [13], journal only metadata and perform in-place writes for data. The two others, NOVA and SplitFS [24], guarantee data-atomicity for each operation. We configured NOVA to use CoW updates, but without enabling checksums. For SplitFS, we configured it to use *strict* mode. We ran PMFS and SplitFS on Linux kernel 4.13, and Ext4-DAX and NOVA on Linux kernel 5.1. Kernel versions are the latest versions that support the underlying file systems.

[5] we measure LOC with sloccount [64]

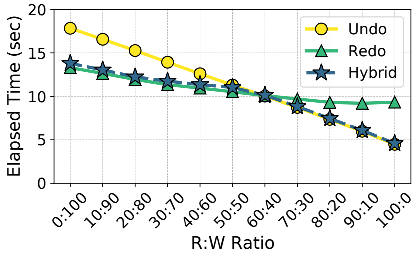


Figure 6: Performance on different logging policies

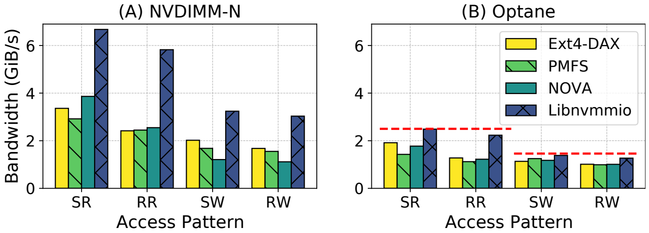


Figure 7: Performance on different access patterns

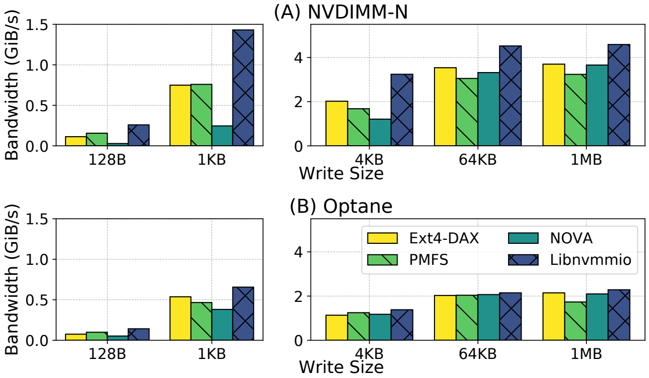


Figure 8: Performance on different write sizes

4.2 Microbenchmark

4.2.1 Hybrid logging

Most logging systems adopt only one logging policy (redo or undo). Each logging policy has different strengths and weaknesses, depending on the type of file accesses. While redo logging is better for write-intensive workloads, undo logging is better for read-intensive workloads.

Figure 6 shows how logging policies (redo, undo, and hybrid logging) affect the performance of Libnvmmio. Undo logging shows better performance than redo, when the workload has high read ratio. Redo logging shows better performance than undo, when the workload has high write ratio. When the *R:W ratio* is 60:40, the two logging policies show the same level of the performance. Based on this observation, Libnvmmio uses the ratio as a change point for its hybrid logging policy. As shown in Figure 6, hybrid logging in Libnvmmio achieves the best case performance of the two logging policies.

4.2.2 Throughput

We measured the bandwidth performance by using FIO [5]. It repeatedly accesses a 4GB file in units of 4KB for 60 seconds in a single thread. Two graphs in Figure 7 show the experiment results on NVDIMM-N (A) and Optane (B), respectively. Four file access patterns are used for our experiment: sequential read (SR), random read (RR), sequential write (SW), and random write (RW). All the other file systems except Libnvmmio perform the file IO at kernel level. Libnvmmio avoids the kernel IO stack overhead and performs file IO mostly at user level.

As shown in Figure 7, Libnvmmio provides the highest throughput on all access patterns, outperforming the other file systems by 1.66$\sim$2.20× on NVDIMM-N and 1.14$\sim$1.74× on Optane. The performance improvements are more noticeable in NVDIMM-N than in Optane. The maximum achievable bandwidths on Optane are 2.5GB/s and 1.46GB/s for FIO mmap based read and write without atomicity support. These are indicated as red dotted lines in Figure 7 (B). The performance results on Optane are almost near the maximum achievable bandwidths for Libnvmmio, which suggests the performance on Optane is limited by the hardware limit, not by the mechanisms in Libnvmmio.

The performance in Libnvmmio is also improved over the other file systems by maximizing logging efficiency in hybrid logging. For read access patterns (SR and RR), Libnvmmio performs only user-level `memcpy` from the memory-mapped file to the user buffer under the undo logging. For write access patterns (SW and RW), Libnvmmio updates only the log, not the memory-mapped file, under the redo logging and asynchronously writes the data from the redo log on `SYNC` call at the file close.

Figure 8 shows the performance of the FIO sequential write on various IO sizes. Libnvmmio performs per-block logging, but provides various log block sizes. With this feature, Libnvmmio can keep the high performance across different IO sizes. The performance generally improves on the increased IO sizes for all file systems and Libnvmmio, as the number of write system calls decreases within the 60 second duration of FIO experiment. Libnvmmio shows significantly higher performance than the other file systems when the IO size is smaller than the page size (128B, 1KB). This is mainly due to the differential logging feature in Libnvmmio. For file systems that use CoW for atomicity, such as NOVA, write amplification becomes a large overhead on sub-page size data writes.

Figure 9 shows the performance of the FIO sequential write on different `fsync` intervals. The horizontal axis represents the fsync frequency. For example, the interval *10* means FIO performed fsync after every ten writes issued. The performance of Ext4-DAX and PMFS slightly increased as the fsync interval increased. Since Ext4-DAX and PMFS perform

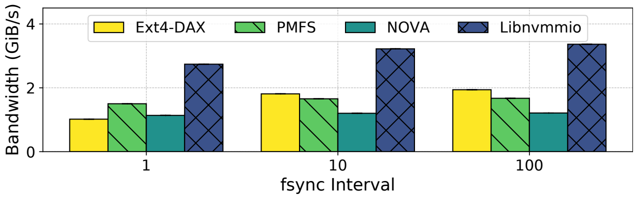


Figure 9: Performance on different `fsync` intervals

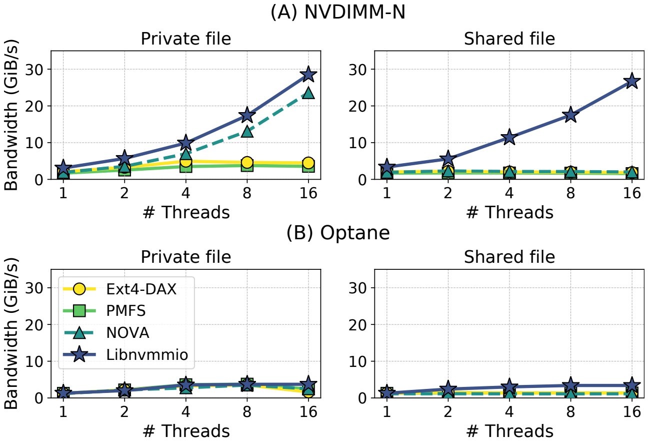


Figure 10: Scalability: FIO random write with multithreads

Latency (us)	Ext4-DAX read	 write	PMFS read	 write	NOVA read	 write	Libnvmmio read	 write
Avg.	1.73	50.43	2.21	6.16	1.73	4.43	1.12	4.14
99th	3	61	3	9	3	9	2	10
99.9th	6	552	4	12	3	10	3	12
99.99th	8	605	8	239	6	15	5	15
99.999th	12	648	17	258	8	5216	7	76

Table 2: 4KB read and write latencies on Optane

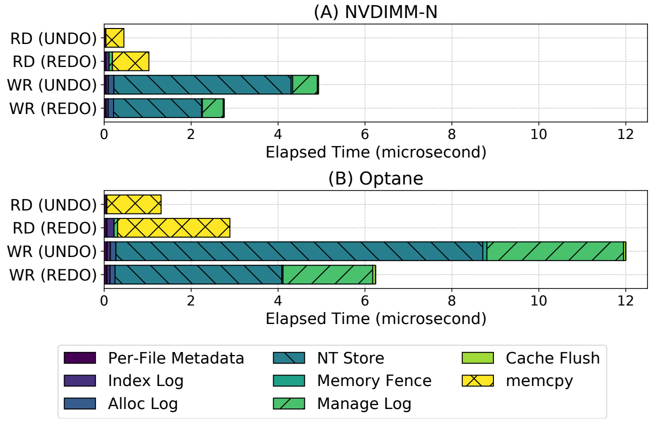


Figure 11: Latency breakdown

only metadata journaling, there is no dramatic performance improvement. NOVA shows the same performance regardless of the fsync interval. Since NOVA performs all the writes atomically and fsync actually does nothing, its performance is not sensitive to the fsync intervals. Libnvmmio implements fsync efficiently with almost little overhead by increasing the current global epoch number at user level. A heavy-lifting work for checkpointing data log is processed in the background. As the fsync interval increases, checkpointing can be done in a batch even in the background. Thus, Libnvmmio can slightly increase the performance on long intervals.

4.2.3 Scalability

Figure 10 shows the performance of multithreaded file IO with FIO random write. In *private file* configuration, each thread writes data to its private file. Whereas, all threads write data to one shared file in *shared file* configuration. In private file configuration on NVDIMM-N, Libnvmmio and NOVA show highly scalable performance. Libnvmmio still shows 29% better performance than NOVA. In contrast, only Libnvmmio sustains scalable performance in shared file configuration on NVDIMM-N. Libnvmmio achieves 13× better performance on 16 threads run than NOVA. It is common for modern applications to access shared files simultaneously from multithreads [40]. While NOVA uses per-inode logging with entire file locks, Libnvmmio uses per-block logging with fine-grained per-block locks. This makes Libnvmmio achieve scalable performance, even when multithreads access the shared file simultaneously. The scalability on Optane is limited mainly due to the memory bandwidth limitation, but Libnvmmio on Optane still shows a little promising results than the others. The other two file systems, Ext4-DAX and PMFS rarely scale on multi-threaded experiments.

4.2.4 Latency

We measured write and read latencies of various NVM-aware file systems and Libnvmmio. To make a fair comparison, all operations are synchronous (`fsync` on every write operation). Table 2 shows the latency of 4KB IO by a single thread. The results were measured on Optane. Libnvmmio outperforms all the other file systems. The advantage of Libnvmmio comes from writing logs in user space and background checkpointing. Ext4-DAX requires copying data between user and kernel buffers, PMFS involves modification of complex data structures, and NOVA requires CoW. Low tail latencies on 99.999th show that Libnvmmio has a high chance to meet the demand for target applications. Since Libnvmmio hooks read/write calls and does not involve any kernel mode switches, Libnvmmio on any file systems can remove the complex techniques the kernel level file systems use. The Libnvmmio latencies on other file systems exhibit almost the same as the ones in Table 2. Our results indicate that applications sensitive to tail latency can adopt Libnvmmio on top of their file systems and drop tail latency dramatically.

Figure 11 shows the latency breakdown of `read` and `write` for two logging policies (undo and redo). As for write, the

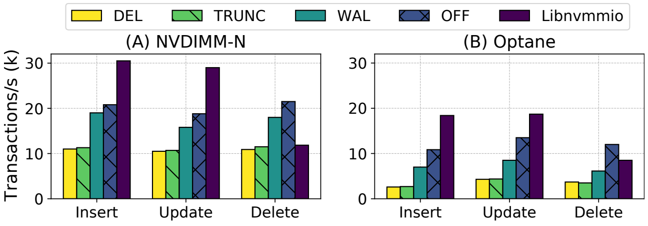


Figure 12: Mobibench on SQLite

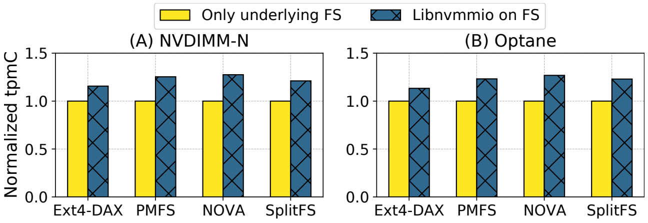


Figure 13: TPC-C on SQLite

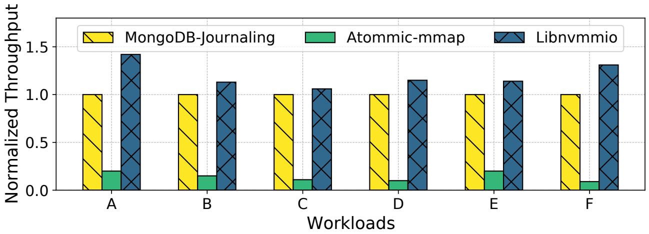


Figure 14: YCSB performance on MongoDB

portion of non-temporal store (NT Store) is dominating. However, the overheads of the memory fence and cache flush is low due to NT store. In this experiment, we confirmed that it is crucial to select an appropriate logging policy according to access types, as the time spent on memory copy (memcpy, NT Store) varies greatly depending on logging policy. The actual seconds for read and write latencies in Figure 11 are bigger than the latency in Table 2, as time measurement routines for breakdown have been injected.

4.3 Real applications

4.3.1 SQLite

We experimented with SQLite [59] to see how Libnvmmio performs in real applications. To guarantee data-atomicity, SQLite uses its own journaling by default. SQLite calls `fsync` on commit to ensure that all data updated in a transaction is persistent. Libnvmmio keeps updated data in its logs and atomically writes to the original file when `fsync` called. This is how data-atomicity can be guaranteed on SQLite on Libnvmmio without the journaling provided by SQLite. However, the file systems we experimented with cannot turn off the journaling. Even file systems that provide data-atomicity for each operation cannot guarantee the atomicity at transaction level without the journaling.

We used Mobibench [41] to evaluate the basic performance of SQLite. In this experiment, we ran SQLite on NOVA with various journal modes: delete (*DEL*), truncate (*TRUNC*), write-ahead logging (*WAL*), no-journaling (*OFF*). Figure 12 shows that Libnvmmio outperforms all journaling modes on insert and update queries. Even when no journaling is provided from SQLite, Libnvmmio outperforms as all file accesses are handled at user level. Compared to WAL mode on NVDIMM-N, insert and update queries have 60% and 93% performance gains in Libnvmmio, respectively. On Optane, the performance gains become 162% and 120%. Mobibench queries request about 100B data IOs. Libnvmmio excels on such small size IOs. On delete transactions, Libnvmmio performs not quite well. According to our call trace, files are truncated frequently on delete workload. When a file is truncated, Libnvmmio needs to adjust the mapping size along with the file size as in FLEX [66] and SplitFS [24]. This incurs relatively high overhead on Libnvmmio. To mend this problem, Libnvmmio needs to optimize file size changes by reflecting file size changes on file close.

We evaluate Libnvmmio on four different file systems by running TPC-C with SQLite. Figure 13 shows that running on Libnvmmio exhibits better performance than running only on underlying file systems. The performance gains range from 16% to 27% on NVDIMM-N and from 13% to 27% on Optane. Since Libnvmmio processes file IO at user level, most of file IO operations can be handled efficiently. As for SplitFS [24], which is built as user-level file system, Libnvmmio uses only mmap interface from SplitFS and performs all other functionalities with its own mechanism. This is why the performance on SplitFS is better for Libnvmmio than only SplitFS. Data updates are kept in its staging files in SplitFS. When applications call `fsync`, SplitFS relinks the updated blocks in staging files into the original file without additional data copying. To make the relink mechanism work, a complete content of the block is required. If applications update only part of a block, SplitFS must copy the rest of the partial data for that block on fsync. The relink mechanism also needs splitting and remapping the existing mapping. Since mapping changes require expensive TLB-shootdown, remapping can cause a higher cost than copying [37]. Additionally, frequent relinks can cause extent fragmentation, as SplitFS uses Ext4-DAX as its underlying file system.

4.3.2 MongoDB MMAPv1

To evaluate Libnvmmio for applications that use memory-mapped IO, we experimented with MongoDB [44] MMAPv1 engine. MongoDB MMAPv1 maps DB files onto its address space, and read/write data with `memcpy`. We have modified 182 lines of source code to make MongoDB MMAPv1 engine use interfaces in Libnvmmio. Figure 14 shows the performance of YCSB workloads on MongoDB. *MongoDB-Journaling* represents the performance when MongoDB uses its own journaling. In order to ensure that all modifications to a MongoDB data set are durably written to DB files, MongoDB, by default, records all modifications to a journal file. After persisting the data in journal, MongoDB writes the data to a memory-mapped file. Then, it calls `msync` periodically to flush the data in the memory to its file image on the persistent storage. If a system failure occurs during the synchronization, MongoDB can redo the updates by using the journal. *Atomic-mmap* represents the performance when MongoDB uses atomic-mmap provided by NOVA [67]. NOVA maps the replica pages of files onto the user memory, and later when `msync` is called, it copies the replica pages atomically to the original file. In this case, MongoDB can guarantee data-atomicity without using its own journaling. Libnvmmio also ensures the same level of data-atomicity as the atomic-mmap in NOVA. *Libnvmmio* represents the performance when Libnvmmio is used without MongoDB journaling. Compared to MongoDB journaling, Libnvmmio shows 31∼42% performance gains on write intensive workloads (A and F). On read intensive workloads (B, C, D, and E), it shows 6∼15% gains.

Libnvmmio shows the highest performance for all workloads. In YCSB workloads, the default record size is 1KB. Since MongoDB-Journaling uses msync provided by the OS kernel, the synchronization is performed at page granularity. This increases the write amplification but also incurs TLB-shootdown overhead. Whereas, Libnvmmio uses differential logging and user-level msync to minimize write amplification and eliminate unnecessary TLB-shootdown. Atomic-mmap also performs synchronization at page granularity. Besides, as all the replica pages of the file are synchronized regardless of their states (clean or dirty), huge write amplification occurs. Due to such inefficiency, the atomic-mmap feature has been removed from the latest NOVA [68].

5 Related Work

In NVMM systems, file operations travel through memory bus led significantly improved latency. In traditional systems, storage latency was dominant in the total file IO overhead, but in NVMM systems, inefficient behavior of software stacks becomes a dominating overhead. State-of-the-art NVMM-aware file systems bypass the block layer and the page cache layer to avoid the software overhead. Many optimizations take the characteristics of NVMM into account in the file system design. Some suggest to fundamentally change the way file operations work from kernel space to user space.

BPFS and PMFS are early versions of NVMM-aware file systems. BPFS [12] manages the CPU cache based on epoch to provide an accurate ordering and provides atomic data persistence with short-circuit shadow paging. PMFS [13] came up with eXecute In Place (XIP) which nowadays call Direct Access (DAX). PMFS pointed out that NVMM systems should bypass the block layer and page cache to remove unnecessary management schemes from past days.

NOVA [67, 68] suggested more efficient software layer to manage NVMM. NOVA extends the log-structuring technique optimized for block devices to NVMM. NOVA gives each inode a separate log. This technique is suited well in NVMM utilizing fast random access characteristics of NVMM. NOVA provides protection against media errors as well as software errors.

Aerie [62] is a user-level file system that provides flexible file system interfaces. Aerie maximizes the benefits of low-latency NVMM by implementing file system functionality at the user-level. However, Aerie does not guarantee data-atomicity and does not support POSIX semantics.

Strata [28] is a cross-media file system that suggested separation of kernel and user responsibilities. While providing fast performance for read and write, Strata does not support atomic memory-mapped IO. Strata brought data into user space and processes metadata in kernel space.

FLEX [66] replaces read/write system calls with memory-mapped IO to avoid entering the OS kernel. FLEX provides transparent user-level file IO, allowing existing applications to utilize the characteristics of NVMM efficiently. However, FLEX does not guarantee data-atomicity.

SplitFS [24] supports user-level IO while providing flexible crash-consistency guarantees. The relink mechanism proposed by SplitFS allows atomic file updates with minimal data copying. SplitFS handles common data operations at the user level and offloads complex and uncommon metadata operations to kernel file systems. SplitFS proposed the proper role of user libraries and kernel file systems for efficient file IO.

6 Conclusion

Libnvmmio is a simple and practical solution, which provides low-latency and scalable IO while guaranteeing data atomicity. Libnvmmio rebuilds performance-critical software IO path for NVM. It leverages the memory-mapped IO for fast data access and makes applications free from the crash-consistency concerns by providing failure-atomicity. Source code is publicly available at: https://github.com/chjs/libnvmmio.

Acknowledgments

This research was supported in part by Samsung Electronics and the National Research Foundation in Korea under PF Class Heterogeneous High Performance Computer Development NRF-2016M3C4A7952587. We would like to thank our shepherd, Ric Wheeler, and the anonymous reviewers for their insightful comments and suggestions.

References

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[2] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. DCS: A Fast and Scalable Device-centric Server Architecture. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48. ACM, 2015.

[3] Nadav Amit. Optimizing the tlb shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 27–39, Berkeley, CA, USA, 2017. USENIX Association.

[4] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2), May 2013.

[5] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[7] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[8] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han. In-memory file system with efficient swap support for mobile smart devices. *IEEE Transactions on Consumer Electronics*, 62(3):275–282, 2016.

[9] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.

[10] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 199–210, New York, NY, USA, 2012. Association for Computing Machinery.

[11] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 211–224, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14. ACM, 2014.

[14] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 353–368, New York, NY, USA, 2016. ACM.

[15] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 9–9, Nov 2005.

[16] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 155–162, New York, NY, USA, 1987. ACM.

[17] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[18] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[19] Intel Optane^{TM} DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Intel Persistent Memory Programming. https://pmem.io/pmdk/.

[21] Intel and Micron's 3D XPoint^{TM} Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[22] Jonathan Corbet. Supporting filesystems in persistent memory, 2014. https://lwn.net/Articles/610174/.

[23] Juchang Lee, Kihong Kim, and S. K. Cha. Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*, pages 173–182, April 2001.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. ACM.

[25] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '16. USENIX Association, 2016.

[26] Hyunjun Kim, Joonwook Ahn, Sungtae Ryu, Jungsik Choi, and Hwansoo Han. In-memory file system for non-volatile memory. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, page 479–484, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16. ACM, 2016.

[28] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association.

[30] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, April 2013.

[31] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM.

[32] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, Jan 2010.

[33] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09. ACM, 2009.

[34] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, New York, NY, USA, 1996. ACM.

[35] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Han, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 1103–1116, New York, NY, USA, 2020. ACM.

[36] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 55–66, New York, NY, USA, 2007. Association for Computing Machinery.

[37] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page

90–103, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Amirsaman Memaripour and Steven Swanson. Breeze : User-Level Access to Non-Volatile Main Memories for Legacy Software. In *2018 IEEE 36st International Conference on Computer Design*, ICCD '18. IEEE, 2018.

[40] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, June 2016. USENIX Association.

[41] Mobibench. https://github.com/ESOS-Lab/Mobibench.

[42] C. Mohan. Repeating history beyond aries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 1–17, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[43] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[44] MongoDB. https://www.mongodb.com.

[45] Netlist NVvault DDR4 NVDIMM-N. https://www.netlist.com/products/specialty-dimms/nvvault-ddr4-nvdimm.

[46] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. ACM, 2016.

[47] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692, New York, NY, USA, 2018. ACM.

[48] Jim Pappas. Annual Update on Interfaces, 2014. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140805_U3_Pappas.pdf.

[49] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, Santa Clara, CA, July 2017. USENIX Association.

[50] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. ACM, 2013.

[51] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies*, FAST '17. USENIX Association, 2017.

[52] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[53] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[54] S. Qiu and A. L. N. Reddy. Exploiting superpages in a nonvolatile memory file system. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, April 2012.

[55] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. . Chen, R. M. Shelby, M. Salinga, D. Krebs, S. . Chen, H. . Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[56] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[57] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10. USENIX Association, 2010.

[58] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Transactions on Storage*, 12(4):19:1–19:27, 2016.

[59] SQLite. https://www.sqlite.org.

[60] Michael M. Swift. Towards o(1) memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17. ACM, 2017.

[61] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349, Oct 2011.

[62] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14. ACM, 2014.

[63] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 357–370, USA, 2013. USENIX Association.

[64] David A. Wheeler. SLOCCount. https://dwheeler.com/sloccount/.

[65] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, 2011.

[66] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies*, FAST '16. USENIX Association, 2016.

[68] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[69] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[70] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies*, FAST '15. USENIX Association, 2015.

[71] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with a Matrix Container in NVM

Ting Yao[1], Yiwen Zhang[1], Jiguang Wan[1*], Qiu Cui[2], Liu Tang[2], Hong Jiang[3],
Changsheng Xie[1], and Xubin He[4]

[1]*WNLO, Huazhong University of Science and Technology, China*
Key Laboratory of Information Storage System, Ministry of Education of China
[2]*PingCAP, China*
[3]*University of Texas at Arlington, USA*
[4]*Temple University, USA*

Abstract

Popular LSM-tree based key-value stores suffer from suboptimal and unpredictable performance due to write amplification and write stalls that cause application performance to periodically drop to nearly zero. Our preliminary experimental studies reveal that (1) write stalls mainly stem from the significantly large amount of data involved in each compaction between L_0-L_1 (i.e., the first two levels of LSM-tree), and (2) write amplification increases with the depth of LSM-trees. Existing works mainly focus on reducing write amplification, while only a couple of them target mitigating write stalls.

In this paper, we exploit non-volatile memory (NVM) to address these two limitations and propose MatrixKV, a new LSM-tree based KV store for systems with multi-tier DRAM-NVM-SSD storage. MatrixKV's design principles include performing smaller and cheaper L_0-L_1 compaction to reduce write stalls while reducing the depth of LSM-trees to mitigate write amplification. To this end, four novel techniques are proposed. First, we relocate and manage the L_0 level in NVM with our proposed *matrix container*. Second, the new *column compaction* is devised to compact L_0 to L_1 at fine-grained key ranges, thus substantially reducing the amount of compaction data. Third, MatrixKV increases the width of each level to decrease the depth of LSM-trees thus mitigating write amplification. Finally, the *cross-row hint search* is introduced for the matrix container to keep adequate read performance. We implement MatrixKV based on RocksDB and evaluate it on a hybrid DRAM/NVM/SSD system using Intel's latest 3D Xpoint NVM device Optane DC PMM. Evaluation results show that, with the same amount of NVM, MatrixKV achieves 5× and 1.9× lower 99^{th} percentile latencies, and 3.6× and 2.6× higher random write throughput than RocksDB and the state-of-art LSM-based KVS NoveLSM respectively.

1 Introduction

Persistent key-value stores are increasingly critical in supporting a large variety of applications in modern data centers. In write-intensive scenarios, log-structured merge trees (LSM-trees) [49] are the backbone index structures for persistent key-value (KV) stores, such as RocksDB [24], LevelDB [25], HBase [26], and Cassandra [35]. Considering that random writes are common in popular OLTP workloads, the performance of random writes, especially sustained and/or bursty random writes, is a serious concern for users [2,41,51]. This paper takes random write performance of KV stores as a major concern. Popular KV stores are deployed on systems with DRAM-SSD storage, which intends to utilize fast DRAM and persistent SSDs to provide high-performance database accesses. However, limitations such as cell sizes, power consumption, cost, and DIMM slot availability prevent the system performance from being further improved via increasing DRAM size [4,23]. Therefore, exploiting non-volatile memories (NVMs) in hybrid systems is widely considered as a promising mechanism to deliver higher system throughput and lower latencies.

LSM-trees [49] store KV items with multiple exponentially increased levels, e.g., from L_0 to L_6. To better understand LSM-tree based KV stores, we experimentally evaluated the popular RocksDB [24] with a conventional system of DRAM-SSD storage, and made observations that point to two challenging issues and their root causes. First, **write stalls** lead to application throughput periodically dropping to nearly zero, resulting in dramatic fluctuations of performance and long-tail latencies, as shown in Figures 2 and 3. The troughs of system throughput indicate write stalls. Write stalls induce highly unpredictable performance and degrade the quality of user experiences, which goes against NoSQL systems' design goal of predictable and stable performance [53,57]. Moreover, write stalls substantially lengthen the latency of request processing, exerting high tail latencies [6]. Our experimental studies demonstrate that the main cause of write stalls is the large amount of data processed in each L_0-L_1 compaction. The L_0-

[*] Corresponding author. Email: jgwan@hust.edu.cn

L_1 compaction involves almost all data in both levels due to the unsorted L_0 (files in L_0 are overlapped with key ranges). The all-to-all compaction takes up CPU cycles and SSD bandwidth, which slows down the foreground requests and results in write stalls and long-tail latency. Second, **write amplification** (WA) degrades system performance and storage devices' endurance. WA is directly related to the depth of the LSM-tree as a deeper tree resulting from a larger dataset increases the number of compactions. Although a large body of research aims at reducing LSM-trees' WA [20,36,41,44,45,51], only a couple of published studies concern mitigating write stalls [6,31,53]. Our study aims to address both challenges simultaneously.

Targeting these two challenges and their root causes, this paper proposes MatrixKV, an LSM-tree based KV store for systems with DRAM-NVM-SSD storage. The design principle behind MatrixKV is leveraging NVM to (1) construct cheaper and finer granularity compaction for L_0 and L_1, and (2) reduce LSM-trees' depth to mitigate WA. The key enabling technologies of MatrixKV are summarized as follows:

Matrix container. The matrix container manages the unsorted L_0 of LSM-trees in NVM with a receiver and a compactor. The receiver adopts and retains the MemTable flushed from DRAM, one MemTable per row. The compactor selects and merges a subset of data from L_0 (with the same key range) to L_1, one column per compaction.

Column compaction. A column compaction is the fine-grained compaction between L_0 and L_1, which compacts a small key range a time. Column compaction reduces write stalls because it processes a limited amount of data and promptly frees up the column in NVM for the receiver to accept data flushed from DRAM.

Reducing LSM-tree depth. MatrixKV increases the size of each LSM-tree level to reduce the number of levels. As a result, MatrixKV reduces write amplification and delivers higher throughput.

Cross-row hint search. MatrixKV gives each key a pointer to logically sort all keys in the matrix container thus accelerating search processes.

2 Background and Motivation

In this section, we present the necessary background on NVM, LSM-trees, LSM-based KV stores, and the challenges and motivations in optimizing LSM-based KV stores with NVMs.

2.1 Non-volatile Memory

Service providers have constantly pursued faster database accesses. They aim at providing users with a better quality of service and experience without a significant increase in the total cost of ownership (TCO). With the emergence and development of new storage media such as phase-change memory [8,33,48,52], memristors [55], 3D XPoint [28], and STT-MRAM [21], enhancing storage systems with NVMs becomes a cost-efficient choice. NVM is byte-addressable, persistent, and fast. It is expected to provide DRAM-like performance, disk-like persistency, and higher capacity than DRAM at a much lower cost [9,16,61]. Compared to SSDs, NVM is expected to provide 100× lower read and write latencies and up to ten times higher bandwidth [3,10,14,22].

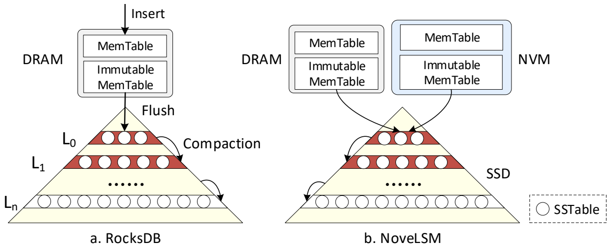


Figure 1: The structure of RocksDB and NoveLSM.

NVM works either as a persistent block storage device accessed through PCIe interfaces or as main memory accessed via memory bus [1,38]. Existing research [31] shows that the former only achieve marginal performance improvements, wasting NVM's high media performance. For the latter, NVM can supplant or complement DRAM as a single-level memory system [27,58,61,65], a system of NVM-SSD [30], or a hybrid system of DRAM-NVM-SSD [31]. In particular, systems with DRAM-NVM-SSD storage are recognized as a promising way to utilize NVMs due to the following three reasons. First, NVM is expected to co-exist with large-capacity SSDs for the next few years [32]. Second, compared to DRAM, NVM still has 5 times lower bandwidth and 3 times higher read latency [28]. Third, a hybrid system balances the TCO and system performance. As a result, MatrixKV focuses on efficiently using NVMs as persistent memory in a hybrid system of DRAM, NVMs, and SSDs.

2.2 Log-structured Merge Trees

LSM-trees [29,49] defer and batch write requests in memory to exploit the high sequential write bandwidth of storage devices. Here we explain a popular implementation of LSM-trees, the widely deployed SSD-based RocksDB [24]. As shown in Figure 1 (a), RocksDB is composed of a DRAM component and an SSD component. It also has a write-ahead log in SSDs protecting data in DRAM from system failures.

To serve write requests, writes are first batched in DRAM by two skip-lists (MemTable and Immutable MemTable). Then, the immutable MemTable is flushed to L_0 on SSDs generating Sorted String Tables (SSTables). To deliver a fast flush, L_0 is unsorted where key ranges overlap among different SSTables. SSTables are compacted from L_0 to deeper levels (L_1, L_2...L_n) during the lifespan of LSM-trees. Compaction makes each level sorted (except L_0) thus bounding the overhead of reads and scans [53].

To conduct a compaction, (1) an SSTable in L_i (called a victim SSTable) and multiple SSTables in L_{i+1} who has overlapping key ranges (called overlapped SSTables) are picked as the compaction data. (2) Other SSTables in L_i that fall in this compaction key ranges are selected reversely. (3) Those SSTables identified in steps (1) and (2) are fetched into memory, to be merged and sorted. (4) The regenerated SSTables are written back to L_{i+1}. Since L_0 is unsorted and each SSTable in L_0 spans a wide key range, the L_0-L_1 compaction performs step (1) and (2) back and forth involving almost all SSTables in both levels, leading to a large all-to-all compaction.

To serve read requests, RocksDB searches the MemTable first, immutable MemTable next, and then SSTables in L_0 through L_n in order. Since SSTables in L_0 contain overlapping keys, a lookup may search multiple files at L_0 [36].

2.3 LSM-tree based KV stores

Existing improvements on LSM-trees includes: reducing write amplification [19, 36, 44, 46, 51, 62–64], improving memory management [7, 39, 56], supporting automatic tuning [17, 18, 40], and using LSM-trees to target hybrid storage hierarchies [5, 47, 50]. Among them, random write performance is a common concern since it is severely hampered by compactions. In the following, we discuss the most related studies of our work in three categories: those reducing write amplification, addressing write stalls, and utilizing NVMs.

Reducing WA: PebblesDB [51] mitigates WA by using guards to maintain partially sorted levels. Lwc-tree [62] provides lightweight compaction by appending data to SSTables and only merging the metadata. WiscKey [36] separates keys from values, which only merges keys during compactions thus reducing WA. The key-value separation solution brings the complexities of garbage collection and range scans and only benefits large values. LSM-trie [59] de-amortizes compaction overhead with hash-range based compaction. VTtree [54] uses an extra layer of indirection to avoid reprocessing sorted data at the cost of fragmentation. TRIAD [44] reduces WA by creating synergy between memory, disk, and log. However, almost all these efforts overlook performance variances and write stalls.

Reducing write stalls: SILK [6] introduces an I/O scheduler which mitigates the impact of write stalls to clients' writes by postponing flushes and compactions to low-load periods, prioritizing flushes and lower level compactions, and preempting compactions. These design choices make SILK exhibits ordinary write stalls on sustained write-intensive and long peak workloads. Blsm [53] proposes a new merge scheduler, called "spring and gear", to coordinate compactions of multiple levels. However, it only bounds the maximum write processing latency while ignoring the large queuing latency [43]. KVell [37] makes KV items unsorted on disks to reduce CPU computation cost thus mitigating write stalls for NVMe SSD based KV stores, which is inapplicable to systems with general SSDs.

Improving LSM-trees with NVMs: SLM-DB [30] proposes a single level LSM-tree for systems with NVM-SSD storage. It uses a B^+-tree in NVM to provide fast read for the single level LSM-tree on SSDs. This solution comes with the overhead of maintaining the consistency between B^+-trees and LSM-trees. MyNVM [23] leverages NVM as a block device to reduce the DRAM usage in SSD based KV stores. NoveLSM [31] is the state-of-art LSM-based KV store for systems with hybrid storage of DRAM, NVMs, and SSDs. NVMRocks [38] aims for an NVM-aware RocksDB, similar to NoveLSM, which adopts persistent mutable MemTables on NVMs. However, as we verified in § 2.4.3, mutable NVM MemTables only reduce access latencies to some extent while generating a negative effect of more severe write stalls.

Since we build MatrixKV for systems with multi-tier DRAM-NVM-SSD storage and redesign LSM-trees to exploit the high performance NVM, NoveLSM [31] is considered the most relevant to our work. We use NoveLSM as our main comparison in evaluations. In addition, we also evaluate PebblesDB and SILK on NVM-based systems since they are state-of-art solutions for reducing WA or write stalls but their original designs are not for the hybrid systems.

2.4 Challenges and Motivations

To explore the challenges in LSM-tree based KV stores, we conduct a preliminary study on the SSD-based RocksDB. In this experiment, an 80 GB dataset of 16bytes-4KB key-value items is written/loaded to RocksDB in uniformly random order. The evaluation environments and other parameters are described in § 5. We record random write throughput every ten seconds as shown in Figure 2. The experimental results expose two challenging issues. **Challenge 1, Write stalls**. System performance experiences peaks and troughs, and the troughs of throughput manifest as write stalls. The significant fluctuations indicate unpredictable and unstable performance. **Challenge 2, Write amplification**. WA causes performance degradation. System performance (i.e., the average throughput) shows a downward trend with the growth of the dataset size since the number of compactions increases with the depth of LSM-trees, bringing more WA.

2.4.1 Write Stalls

In an LSM-based KV store, there are three types of possible stalls as depicted in Figure 1(a). (1) Insert stalls: if MemTable fills up before the completion of background flushes, all insert operations to LSM-trees are stalled [31]. (2) Flush stalls: if L_0 has too many SSTables and reaches a size limit, flushes to

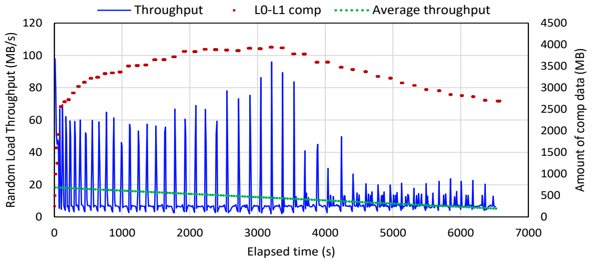


Figure 2: RocksDB's random write performance and L_0-L_1 compactions. *The blue line shows the random write throughput measured in every 10 seconds. The green line shows the average throughput. Each red line represents the duration and amount of data processed in a L_0-L_1 compaction.*

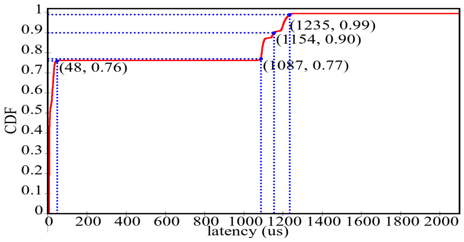


Figure 3: The CDF of latencies of the 80 GB write requests.

storage are blocked. (3) Compaction stalls: too many pending compaction bytes block foreground operations. All these stalls have a cascading impact on write performance and result in write stalls.

Evaluating these three types of stalls individually by recording the period of flushes and compactions at different levels, we find that the period of L_0-L_1 compaction approximately matches write stalls observed, as shown in Figure 2. Each red line represents a L_0-L_1 compaction, where the length along the x-axis represents the latency of the compaction and the right y-axis shows the amount of data processed in the compaction. The average amount of compaction data is 3.10 GB. As we elaborate in § 2.2, since L_0 allows overlapping key ranges between SSTables, almost all SSTables in both levels join the L_0-L_1 compaction. A large amount of compaction data leads to heavy read-merge-writes, which takes up CPU cycles and the SSD bandwidth, thus blocking foreground requests and making L_0-L_1 compaction the primary cause of write stalls.

Write stalls not only are responsible for the low system throughput, but also induce high write latency leading to the long-tail latency problem. Figure 3 shows the cumulative distribution function (CDF) of the latency for each write request during the 80 GB random load process. Although the latency of 76% of the write requests is less than 48 us, the write latency of the 90^{th}, 99^{th}, and 99.9^{th} percentile reaches 1.15, 1.24, and 2.32 ms respectively, a two-order magnitude increase. The high latency significantly degrades the quality of user experiences, especially for latency-critical applications.

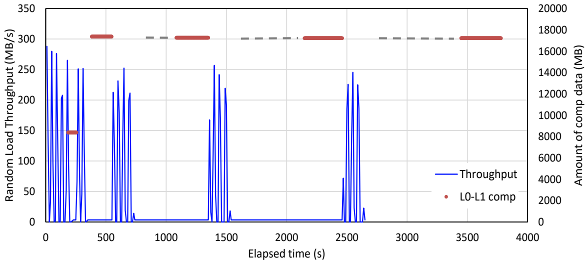


Figure 4: NoveLSM's random write performance and L_0-L_1 compactions. *Comparing to RocksDB in Figure 2, the average period of write stalls is increased.*

2.4.2 Write Amplification

Next, we analyze the second observation, i.e., system throughput degrades with the increase in dataset size. Write amplification (WA) is defined as the ratio between the amount of data written to storage devices and the amount of data written by users. LSM-tree based KV stores have long been criticized for their high WA due to frequent compactions. Since the sizes of adjacent levels from low to high increase exponentially by an amplification factor ($AF = 10$), compacting an SSTable from L_i to L_{i+1} results in a WA factor of AF on average. The growing size of the dataset increases the depth of an LSM-tree as well as the overall WA. For example, the WA factor of compacting from L_1 to L_2 is AF, while the WA factor of compacting from L_1 to L_6 is over $5 \times AF$. The increased WA consumes more storage bandwidth, competes with flush operations, and ultimately slows down application throughput. Hence, system throughput decreases with higher write amplification caused by the increased depth of LSM-trees.

2.4.3 NoveLSM

NoveLSM [31] exploits NVMs to deliver high throughput for systems with DRAM-NVM-SSD storage, as shown in Figure 1(b). The design choices of NoveLSM include: (1) adopting NVMs as an alternative DRAM to increase the size of MemTable and immutable MemTable; (2) making the NVM MemTable mutable to allow direct updates thus reducing compactions. However, these design choices merely postpone the write stalls. When the dataset size exceeds the capacity of NVM MemTables, flush stalls still happen, blocking foreground requests. Furthermore, the enlarged MemTables in NVM are flushed to L_0 and dramatically increase the amount of data in L_0-L_1 compactions, resulting in even more severe write stalls. The worse write stalls magnify performance variances and hurt user experiences further.

We evaluate NoveLSM (with 8 GB NVM) by randomly writing the same 80 GB dataset. Test results in Figure 4 show that NoveLSM reduces the overall loading time by 1.7× compared to RocksDB (Figure 2). However, the period of write stalls is significantly longer. This is because the amount

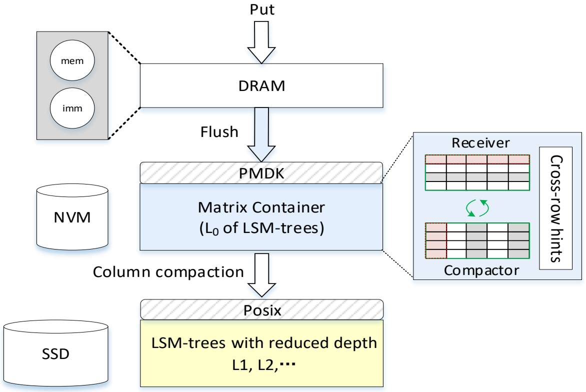


Figure 5: MatrixKV's architectural overview. *MatrixKV is a KV store for systems consisting of DRAM, NVMs, and SSDs.*

of data involved in each L_0-L_1 compaction is over 15 GB, which is 4.86× larger than that of RocksDB. A write stall starts when compaction threads call for the L_0-L_1 compaction. Then, the compaction waits and starts until other pending compactions with higher priorities complete (i.e., the grey dashed lines). Finally, performance rises again as the compaction completes. In general, NoveLSM exacerbates write stalls.

From the above analysis, we conclude that the main cause of write stalls is the large amount of data involved in L_0-L_1 compactions, and the main cause of increased WA is the deepened depth of LSM-trees. The compounded impact of write stalls and WA deteriorates system throughput and lengthens tail latency. While NoveLSM attempts to alleviate these issues, it actually exacerbates the problem of write stalls. Motivated by these observed challenging issues, we propose MatrixKV that aims at providing a stable low-latency KV store via intelligent use of NVMs, as elaborated in the next section.

3 MatrixKV Design

In this section, we present MatrixKV, an LSM-tree based key-value store for systems with multi-tier DRAM-NVM-SSD storage. MatrixKV aims to provide predictable high performance through the efficient use of NVMs with the following four key techniques, i.e., the matrix container in NVMs to manage the L_0 of LSM-trees (§ 3.1), column compactions for L_0 and L_1 (§ 3.2), reducing LSM-tree levels (§ 3.3), and the cross-row hint search (§ 3.4). Figure 5 shows the overall architecture of MatrixKV. From top to bottom, (1) DRAM batches writes with MemTables, (2) MemTables are flushed to L_0 that is stored and managed by the matrix container in NVMs, (3) data in L_0 are compacted to L_1 in SSDs through column compactions, and (4) SSDs store the remaining levels of a flattened LSM-tree.

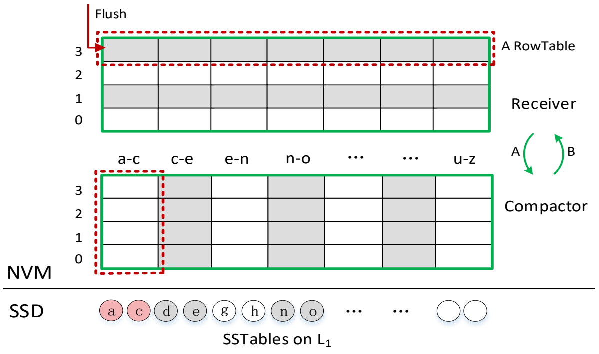


Figure 6: Structure of matrix container. *The receiver absorbs flushed MemTables, one per row. Each row is reorganized as a RowTable. The compactor merges L_0 with L_1 in fine-grained key ranges, one range at a time, referred to as* column compaction. *In Process A, the receiver becomes the compactor once RowTables fill its capacity. In Process B, each column compaction frees a column.*

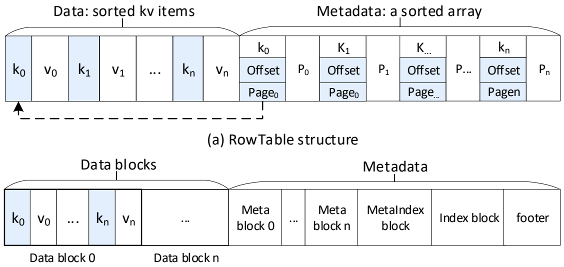
3.1 Matrix Container

LSM-tree renders all-to-all compactions for L_0 and L_1 because L_0 has overlapping key ranges among SSTables. The heavy L_0-L_1 compactions are identified as the root cause of write stalls as demonstrated in § 2.4. NoveLSM [31] exploits NVM to increase the number and size of MemTables. However, it actually exacerbates write stalls by having a larger L_0 and keeping the system bottleneck, L_0-L_1 compactions, on lower-speed SSDs. Hence, the principle of building an LSM-tree based KV store without write stalls is to reduce the granularity of L_0-L_1 compaction via high-speed NVMs.

Based on this design principle, MatrixKV elevates L_0 from SSDs to NVMs and reorganizes L_0 into a matrix container to exploit the byte-addressability and fast random accesses of NVMs. Matrix container is a data management structure for the L_0 of LSM-trees. Figure 6 shows the organization of a matrix container, which comprises one receiver and one compactor.

Receiver: In the matrix container, the receiver accepts and retains MemTables flushed from DRAM. Each such MemTable is serialized as a single row of the receiver and organized as a *RowTable*. RowTables are appended to the matrix container row by row with an increasing sequence number, i.e., from 0 to n. The size of the receiver starts with one RowTable. When the receiver size reaches its size limit (e.g., 60% of the matrix container) and the compactor is empty, the receiver stops receiving flushed MemTables and dynamically turns into the compactor. In the meantime, a new receiver is created for receiving flushed MemTables. There is no data migration for the logical role change of the receiver to the compactor.

RowTable: Figure 7(a) shows the RowTable structure consisting of data and metadata. To construct a RowTable, we first serialize KV items from the immutable MemTable in the



(a) RowTable structure

(b) Conventional SSTable structure

Figure 7: RowTable and conventional SSTable.

order of keys (the same as SSTables) and store them to the data region. Then, we build the metadata for all KV items with a sorted array. Each array element maintains the key, the page number, the offset in the page, and a forward pointer (i.e., p_n). To locate a KV item in a RowTable, we binary search the sorted array to get the target key and find its value with the page number and the offset. The forward pointer in each array element is used for cross-row hint searches that contribute to improving the read efficiency within the matrix container. The cross-row hint search will be discussed in § 3.4. Figure 7(b) shows the structure of conventional SSTable in LSM-trees. SSTables are organized with the basic unit of blocks in accordance with the storage unit of devices such as SSDs and HDDs. Instead, RowTable takes an NVM page as its basic unit. Other than that, RowTables are only different from SSTables in the organization of metadata. As a result, the construction overhead of SSTables and RowTables is similar.

Compactor: The compactor is used for selecting and merging data from L_0 to L_1 in SSDs at a fine granularity. Leveraging the byte addressability of NVMs and our proposed RowTables, MatrixKV allows cheaper compactions that merge a specific key range from L_0 with a subset of SSTables at L_1 without needing to merge all of L_0 and all of L_1. This new L_0-L_1 compaction is referred to as column compaction (detailed in § 3.2). In the compactor, KV items are managed by logical columns. A column is a subset of key spaces with a limited amount of data, which is the basic unit of the compactor in column compactions. Specifically, KV items from different RowTables that fall in the key range of a column compaction logically constitute a column. The amount of these KV items is the size of a column, which is not strictly fixed but at a threshold determined by the size of column compactions.

Space management: After compacting a column, the NVM space occupied by the column is freed. To manage those freed spaces, we simply apply the paging algorithm [3]. Since column compactions rotate the key ranges, at most one page per RowTable is partially fragmented. The NVM pages fully freed after column compactions are added to the free list as a group of page-sized units. To store incoming RowTables in the receiver, we apply free pages from the free list. The 8 GB matrix container contains 2^{11} pages of 4 KB each. Each page is identified by the page number of an unsigned integer. Adding the 8 bytes pointer per list element, the metadata size for each page is 12 bytes. The metadata of the free list occupies a total space of 24 KB on NVMs at most.

It is worth noting that in the matrix container, while columns are being compacted in the compactor, the receiver can continue accepting flushed MemTables from DRAM simultaneously. By freeing the NVM space one column at a time, MatrixKV ends the write stalls forced by merging the entire L_0 with all of L_1.

3.2 Column Compaction

Column compaction is a fine-grained L_0-L_1 compaction that each time compacts only a column, i.e., a small subset of the data in a specific key range. Thus, column compaction can significantly reduce write stalls. The main workflow of column compaction can be described in the following seven steps. (1) MatrixKV separates the key space of L_1 into multiple contiguous key ranges. Since SSTables in L_1 are sorted and each SSTable is bounded by its smallest key and largest key, the smallest keys and largest keys of all the SSTables in L_1 form a sorted key list. Every two adjacent keys represent a key range, i.e., the key range of an SSTable or the gap between two adjacent SSTables. As a result, we have multiple contiguous key ranges in L_1. (2) Column compaction starts from the first key range in L_1. It selects a key range in L_1 as the compaction key range. (3) In the compactor, victim KV items within the compaction key range are picked concurrently in multiple rows. Specifically, assuming N RowTables in the compactor, k threads work in parallel to fetch keys within the compaction key range. Each thread in charge of N/k RowTables. We maintain an adequate degree of concurrent accesses on NVMs with $k = 8$. (4) If the amount of data within this key range is under the lower bound of compaction, the next key range in L_1 joins. The k threads keep forward in N sorted arrays (i.e., the metadata of the RowTables) fetching KV items within the new key range. This key range expansion process continues until the amount of compaction data reaches a size between the lower bound and the upper bound (i.e., $\frac{1}{2}AF \times S_{sst}$ and $AF \times S_{sst}$ respectively). The two bounds guarantee the adequate overhead of a column compaction. (5) Then a column in the compactor is logically formed, i.e., KV items in N RowTables that fall in the compaction key range make up a logical column. (6) Data in the column are merged and sorted with the overlapped SSTables of L_1 in memory. (7) Finally, the regenerated SSTables are written back to L_1 on SSDs. Column compaction continues between the next key range of L_1 and the next column in the compactor. The key ranges of column compaction rotate in the whole key space to keep LSM-trees balanced.

We show an example of column compaction in Figure 8. First, MatrixKV picks the SSTable with key range 0-3 in L_1 as the candidate compaction SSTable. Then, we search the metadata arrays of the four RowTables. If the amount

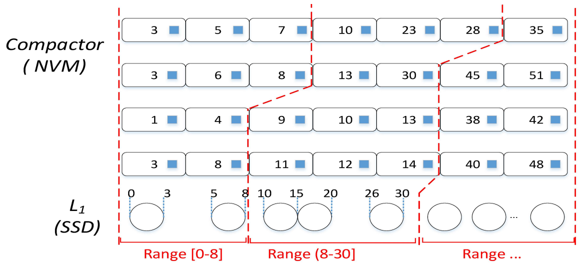


Figure 8: Column compaction: an example. *There are 4 RowTables in the compactor. Each circle represents an SSTable on L_1. Columns are logically divided (red dashed lines) according to the key range of compaction.*

of compaction data within key range 0-3 is under the lower bound, the next key range (i.e., key range 3-5) joins to form a larger key range 0-5. If the amount of compaction data is still beneath the lower bound, the next key range 5-8 joins. Once the compaction data is larger than the lower bound, a logical column is formed for the compaction. The first column compaction compacts the column at the key range of 0-8 with the first two SSTables in L_1.

In general, column compaction first selects a specific key range from L_1, and then compacts with the column in the compactor that shares the same key range. Comparing to the original all-to-all compaction between L_0 and L_1, column compaction compacts at a much smaller key range with a limited amount of data. Consequently, the fine-grained column compaction shortens the compaction duration, resulting in reduced write stalls.

3.3 Reducing LSM-tree Depth

In conventional LSM-trees, the size limit of each level grows by an amplification factor of $AF = 10$. The number of levels in an LSM-tree increase with the amount of data in the database. Since compacting an SSTable to a higher level results in a write amplification factor of AF, the overall WA increases with the number of levels (n) in the LSM-tree, i.e., WA=n*AF [36]. Hence, the other design principle of MatrixKV is to reduce the depth of LSM-trees to mitigate WA. MatrixKV reduces the number of LSM-tree levels by increasing the size limit of each level at a fixed ratio making the AF of adjacent levels unchanged. As a result, for compactions from L_1 and higher levels, the WA of compacting an SSTable to the next level remains the same AF but the overall WA is reduced with due to fewer levels.

Flattening conventional LSM-trees with wider levels brings two negative effects. First, since the enlarged L_0 has more SSTables that overlap with key ranges, the amount of data in each L_0-L_1 compaction increases significantly, which not only adds the compaction overhead but also lengthens the duration of write stalls. Second, traversing the larger unsorted L_0 decreases the search efficiency. MatrixKV addresses the first problem with the fine-grained column compaction. The amount of data involved in each column compaction is largely independent of the level width as a column contains a limited amount of data. For the second problem, MatrixKV proposes the cross-row hint search (§ 3.4) to compensate for the increased search overhead due to the enlarged L_0. It is worth noting that locating keys in fewer levels reduces the lookup time on SSDs, since SSTables from L_1 to L_n are well-sorted.

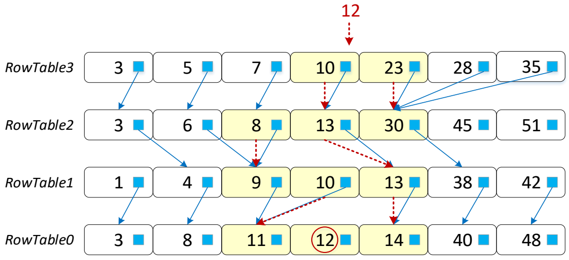


Figure 9: Cross-row hint search. *This figure shows an example of searching the target key ($k = 12$) with forward pointers of each array element.*

3.4 Cross-row Hint Search

In this section, we discuss solutions for improving the read efficiency in the matrix container. In the L_0 of MatrixKV, each RowTable is sorted and different RowTables are overlapped with key ranges. Building Bloom filters for each table is a possible solution for reducing search overheads. However, it brings costs on the building process and exhibits no benefit to range scans. To provide adequate read and scan performances for MatrixKV, we build cross-row hint searches.

Constructing cross-row hints: When we build a RowTable for the receiver of the matrix container, we add a forward pointer for each element in the sorted array of metadata (Figure 7). Specifically, for a key x in RowTable i, the forward pointer indexes the key y in the preceding RowTable $i-1$, where the key y is the first key not less than x (i.e., $y \geq x$). These forward pointers provide hints to logically sort all keys in different rows, similar to the fractional cascading [11, 53]. Since each forward pointer only records the array index of the preceding RowTable, the size of a forward pointer is only 4 bytes. Thus, the storage overhead is very small.

Search process in the matrix container: A search process starts from the latest arrived RowTable i. If the key range of RowTable i does not overlap the target key, we skip to its preceding RowTable $i-1$. Else, we binary search RowTable i to find the key range (i.e., bounded by two adjacent keys) where the target key resides. With the forward pointers, we can narrow the search region in prior RowTables, $i-1, i-2, \ldots$ continually until the key is found. As a result, there is no need to traverse all tables entirely to get a key or scan a key range.

Cross-row hint search improves the read efficiency of L_0 by significantly reducing the number of tables and elements involved in a search process.

An example of cross-row hint search is shown in Figure 9. The blue arrows show the forward pointers providing cross-row hints. Suppose we want to fetch a target key $k = 12$ in the matrix container, we first binary search RowTable 3 to get a narrowed key range of key=10 to key=23. Then their hints lead us to the key 13 and 30 in RowTable 2 (the red arrows). The preceding key is added into the search region when the target key is not included in the key range of the two hint keys. Next, we binary search between key=8 and key=30. Failing to find the target key, we move to the prior RowTable 1, then RowTable 0, with the forward pointers. Finally, the target key 12 is obtained in RowTable 0.

4 Implementation

We implement MatrixKV based on the popular KV engine RocksDB [24] from Facebook. The LOC on top of RocksDB is 4117 lines [1]. As shown in Figure 5, MatrixKV accesses NVMs via the PMDK library and accesses SSDs via the POSIX API. The persistent memory development kit (PMDK) [1, 60] is a library based on the direct access feature (DAX). Next, we briefly introduce the write and read processes and the mechanism for consistency as follows.

Write: (1) Write requests from users are inserted into a write-ahead log on NVMs to prevent data loss from system failures. (2) Data are batched in DRAM, forming MemTable and immutable MemTable. (3) The immutable MemTable is flushed to NVM and stored as a RowTable in the receiver of the matrix container. (4) The receiver turns into the compactor logically if the number of RowTables reaches a size limit (e.g., 60% of the matrix container) and the compactor is empty. This role change has no real data migrations. (5) Data in the compactor is column compacted with SSTables in L_1 column by column. In the meantime, a new receiver receives flushed MemTables. (6) In SSDs, SSTables are merged to higher levels via conventional compactions as RocksDB does. Compared to RocksDB, MatrixKV is completely different from step 3 through step 5.

Read: MatrixKV processes read requests in the same way as RocksDB. The read thread searches with the priority of DRAM>NVMs>SSDs. In NVMs, the cross-row hint search contributes to faster searches among different RowTables of L_0. The read performance can be further improved by concurrently searching in different storage devices [31].

Consistency: Data structures in NVM must avoid inconsistency caused by system failures [12, 13, 34, 42, 58]. For MatrixKV, writes/updates for NVM only happen in two processes, flush and column compaction. For flush, immutable MemTables flushed from DRAM are organized as RowTables and written to NVM in rows. If a failure occurs in the middle of writing a RowTable, MatrixKV can re-process all the transactions that were recorded in the write-ahead log. For column compaction, MatrixKV needs to update the state of RowTables after each column compaction. To achieve consistency and reliability with low overhead, MatrixKV adopts the versioning mechanism of RocksDB. RocksDB records the database state with a manifest file. The operations of compaction are persisted in the manifest file as version changes. If the system crashes during compaction, the database goes back to its last consistent state with versioning. MatrixKV adds the state of RowTables into the manifest file, i.e., the offset of the first key, the number of keys, the file size, and the metadata size, etc. MatrixKV uses lazy deletion to guarantee that stale columns invalidated by column compactions are not deleted until a consistent new version is completed.

Table 1: FIO 4 KB read and write bandwidth

	SSDSC2BB800G7	Optane DC PMM
Rnd write	68 MB/s	1363 MB/s
Rnd read	250 MB/s	2346 MB/s
Seq write	354 MB/s	1444 MB/s
Seq read	445 MB/s	2567 MB/s

5 Evaluation

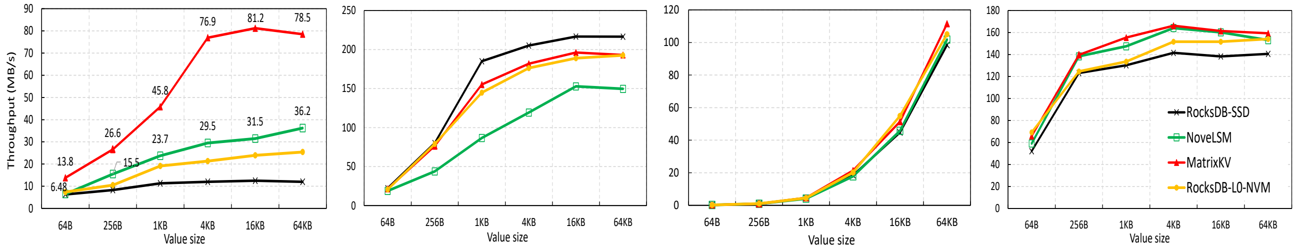
In this section, we run extensive experiments to demonstrate the key accomplishments of MatrixKV. (1) MatrixKV obtains better performance on various types of workloads and achieves lower tail latencies (§ 5.2). (2) The performance benefits of MatrixKV come from reducing write stalls and write amplification by its key enabling techniques (§ 5.3).

5.1 Experiment Setup

All experiments are run on a test machine with two Genuine Intel(R) 2.20GHz 24-core processors and 32 GB of memory. The kernel version is 64-bit Linux 4.13.9 and the operating system in use is Fedora 27. The experiments use two storage devices, an 800 GB Intel SSDSC2BB800G7 SSD and 256 GB NVMs of two 128 GB Intel Optane DC PMM [28]. Table 1 lists their maximum single-thread bandwidth, evaluated with the versatile storage benchmark tool FIO.

We mainly compare MatrixKV with NoveLSM and RocksDB (including RocksDB-SSD and RocksDB-L0-NVM). RocksDB-SSD represents the conventional RocksDB on a DRAM-SSD hierarchy. The other three KV stores are for systems with DRAM-NVM-SSD storage. They use 8 GB NVM to be consistent with the setup in NoveLSM's paper and force the majority of the 80 GB test data to be flushed to SSDs. RocksDB-L0-NVM simply enlarges L_0 into 8 GB and stores it in NVM. MatrixKV reorganizes the 8 GB L_0

[1] MatrixKV source code is publicly available at https://github.com/PDS-Lab/MatrixKV.



(a) Random write

(b) Sequential write

(c) Random read

(d) Sequential read

Figure 10: Performance on Micro-benchmarks with different value sizes.

in NVM and enlarges the L_1 in SSDs into the same 8 GB. NoveLSM employs NVM to store two MemTables (2*4 GB). Test results from this configuration can also demonstrate that MatrixKV achieves system performance improvement with the economical use of NVMs. Finally, we evaluate PebblesDB and SILK for systems with DRAM-NVM storage since they are the representative studies on LSM-tree improvement but are not originally designed for systems with multi-tier storage. Unless specified otherwise, the evaluated KV stores assume the default configuration of RocksDB, i.e., 64 MB MemTables/SSTables, 256 MB L_1 size, and AF of 10. The default key-value sizes are 16 bytes and 4 KB.

5.2 Overall performance evaluation

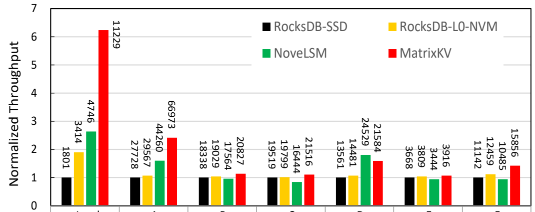
In this section, we first evaluate the overall performance of the four KV stores using db_bench, the micro-benchmark released with RocksDB. Then, we evaluate the performance of each KV store with the YCSB macro-benchmarks [15].

Write performance: We evaluate the random write performance by inserting KV items totaling 80 GB in a uniformly distributed random order. Figure 10(a) shows the random write throughput of four KV stores as a function of value size. The performance difference between RocksDB-SSD and RocksDB-L0-NVM suggests that simply placing L_0 in NVM brings about an average improvement of 65%. We use RocksDB-L0-NVM and NoveLSM as baselines of our evaluation. MatrixKV improves random write throughput over RocksDB-L0-NVM and NoveLSM in all value sizes. Specifically, MatrixKV's throughput improvement over RocksDB-L0-NVM ranges from 1.86× to 3.61×, and MatrixKV's throughput improvement over NoveLSM ranges from 1.72× to 2.61. Taking the commonly used value size of 4 KB as an example, MatrixKV outperforms RocksDB-L0-NVM and NoveLSM by 3.6× and 2.6× respectively. RocksDB-L0-NVM delivers relatively poor performance since putting L_0 in NVM only brings a marginal improvement. NoveLSM uses a large mutable MemTable in NVM to handle a portion of update requests thus slightly reducing WA. However, for both RocksDB and NoveLSM, the root causes of write stalls and WA remain unaddressed, i.e., the all-to-all L_0-L_1 compaction and the deepened depth of LSM-trees.

We evaluate sequential write performance by inserting a total of 80 GB KV items in sequential order. From the test results in Figure 10(b), we make three main observations. First, sequential write throughput is higher than random write throughput for the four KV stores as sequential writes incur no compaction. Second, RocksDB-SSD performs the best since the other three KV stores have an extra NVM tier, requiring data migration from NVMs to SSDs. Three, MatrixKV and RocksDB-L0-NVM have better sequential write throughput than NoveLSM since contracting RowTable/SSTables in NVMs is cheaper than updating the skip list of NoveLSM's large mutable MemTable.

Read performance: Random/sequential read performances are evaluated by reading one million KV items from the 80 GB randomly loaded database. To obtain the read performance free from the impact of compactions, we start the reading test after the tree becomes well-balanced. Figure 10(c) and (d) show the test results of random reads and sequential reads. Since NVM only accommodates 10% of the dataset, the read performance in SSDs dominates the overall read performance. Besides, since a balanced tree is well-sorted from L_1 to L_n on SSDs, the four KV stores exhibit similar read throughputs. MatrixKV does not degrade read performance and even has a slight advantage in sequential reads for two reasons. First, the cross-row hint search reduces the search overhead of the enlarged L_0. Second, MatrixKV has fewer LSM-tree levels, resulting in less search overhead on SSDs.

Macro-benchmarks: Now we evaluate four KV stores with YCSB [15], a widely used macro-benchmark suite delivered by Yahoo!. We first write an 80 GB dataset with 4KB values for loading, then evaluate workload A-F with one million KV items respectively. From the test results shown in Figure 11, we draw three main conclusions. First, MatrixKV gets the most advantage from write/load dominated workloads, i.e., load, and workload A and F. MatrixKV is 3.29× and 2.37× faster than RocksDB-L0-NVM and NoveLSM on the load workload (i.e., random write). Second, MatrixKV maintains adequate performance over read-dominated workloads, i.e., workloads B to E. Third, NoveLSM and MatrixKV behave better on workload D due to the latest distribution, where they both hit more in NVMs and thus MatrixKV can



	Load	A	B	C	D	E	F
Inserts	100%	0%	0%	0%	5%	5%	0%
Updates	0%	50%	5%	0%	0%	0%	0%
Reads	0%	50%	95%	100%	95%	0%	50%
Range query	0%	0%	0%	0%	0%	95%	0%
RMW	0%	0%	0%	0%	0%	0%	50%
Distribution	Uniform	Zipfian	Zipfian	Zipfian	Latest	Zipfian	Zipfian

Figure 11: Macro-benchmarks. *The y-axis shows the throughput of each KV store normalized to RocksDB-SSD. The number on each bar indicates the throughput in ops/s.*

Table 2: Tail Latency

Latency (us)	avg.	90%	99%	99.9%
RocksDB-SSD	974	566	11055	17983
NoveLSM	450	317	2080	2169
RocksDB-L0-NVM	477	528	786	1112
MatrixKV	**263**	**247**	**405**	**663**

benefit more from cross-row hints.

Tail latency: Tail latency is especially important for LSM-tree based KV stores, since they are widely deployed in production environments to provide services for write-heavy workloads and latency-critical applications. We evaluate the tail latency with the same methodology used in SILK [6], i.e., using the YCSB-A workload and setting request arrival rate at around 20K requests/s. Table 2 shows the average, 90^{th}, 99^{th}, and 99.9^{th} percentile latencies of four key-value stores. MatrixKV significantly reduces latencies in all cases. The 99^{th} percentile latency of MatrixKV is 27×, 5×, and 1.9× lower than RocksDB-SSD, NoveLSM, and RocksDB-L0-NVM respectively. The test results demonstrate that by reducing write stalls and WA, MatrixKV improves the quality of user experience with much lower tail latencies.

5.3 Performance Gain Analysis

To understand MatrixKV's performance improvement over random write workloads, we investigate the main challenges of LSM-trees (§ 5.3.1) and the key enabling techniques of MatrixKV (§ 5.3.2).

5.3.1 Main Challenges

In this section, we demonstrate that MatrixKV does address the main challenges of LSM-trees, i.e., write stalls and WA.

Write Stalls: We record the throughput of the four KV stores in every ten seconds during their 80 GB random write process (similar to Figures 2 and 4) to visualize write stalls. From the performance variances shown in Figure 12, we draw three observations. (1) MatrixKV takes a shorter time to process the same 80GB random write since it has higher random write throughput than other KV stores (as demonstrated in § 5.2). (2) Both RocksDB and NoveLSM suffer from write stalls due to the expensive L_0-L_1 compaction. NoveLSM takes longer to process a L_0-L_1 compaction because L_0 maintains large MemTables flushed from NVMs. Comparing to RocksDB-SSD, RocksDB-L0-NVM has lower throughput during write stalls, which means that it blocks foreground requests more severely because of the enlarged L_0. (3) MatrixKV achieves the most stable performance. The reason is that we reduce write stalls by the fine-grained column compaction which guarantees a small amount of data processed in each L_0-L_1 compaction.

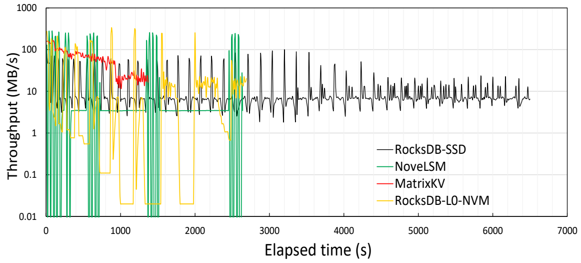


Figure 12: Throughput fluctuation as a function of time. *The random write performance fluctuates where the troughs on curves signify the occurrences of possible write stalls.*

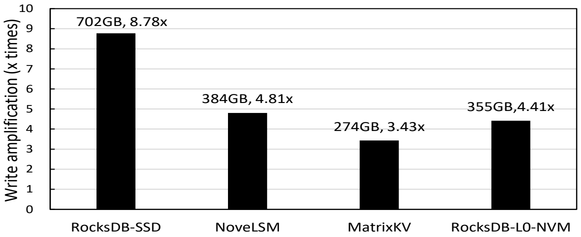


Figure 13: Write amplification of 80 GB random writes.*The numbers on each bar show the amount of data written to SSDs and the WA ratio respectively.*

Write Amplification: We measure the WA of four systems on the same experiment of randomly writing 80 GB dataset. Figure 13 shows the WA factor measured by the ratio of the amount of data written to SSDs and the amount of data coming from users. The WA of MatrixKV, NoveLSM, and RocksDB-L0-NVM are 2.56×, 1.83×, and 1.99× lower than RocksDB-SSD respectively. MatrixKV has the smallest WA since it reduces the number of compactions by lowering the depth of LSM-trees.

5.3.2 MatrixKV Enabling Techniques

Column compaction: To demonstrate the efficiency of column compaction, we record the amount of data involved, the duration of every L_0-L_1 compaction for four KV stores in the same 80 GB random write experiment. As shown in Figure 14, MatrixKV conducts 467 column compactions, each 0.33 GB,

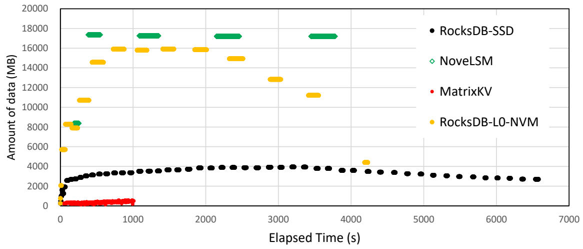


Figure 14: The L_0-L_1 compaction. *Each line segment indicates an L_0-L_1 compaction. The y-axis shows the amount of data involved in the compaction and the length along x-axis shows the duration of the compaction.*

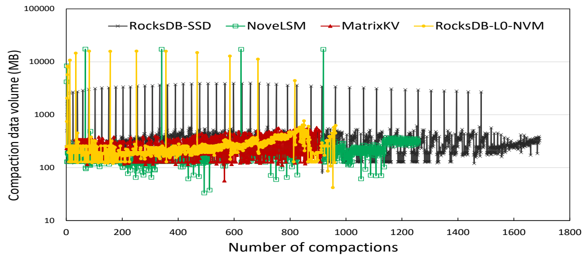
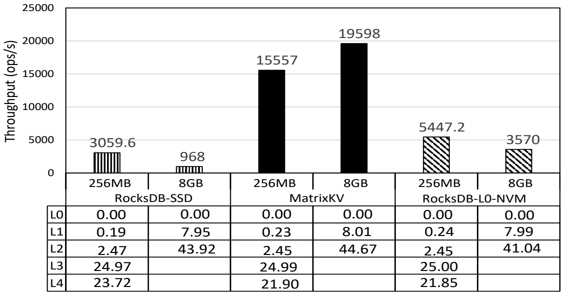


Figure 15: Compaction analysis. *This figure shows the amount of data of every individual compaction during the 80 GB random write.*

written a total of 153 GB data. RocksDB-SSD processes 52 compactions, each 3.1 GB on average, written a total of 157 GB data. MatrixKV processes more fine-grained L_0-L_1 compactions, where each has the least amount of data and the shortest compaction duration. As a result, column compactions have only a negligible influence on foreground requests and finally significantly reduce write stalls. NoveLSM actually exacerbates write stalls since the enlarged MemTables flushed from NVM significantly increase the amount of data processed in each L_0-L_1 compaction.

Overall compaction efficiency: We further record the overall compaction behaviors of four KV stores by recording the amount of data for every compaction during the random write experiment. From the test results shown in Figure 15, we draw four observations. First, MatrixKV has the smallest number of compactions, attributed to the reduced LSM-tree depth. Second, all compactions in MatrixKV process similar amount of data since we reduce the amount of compaction data on L_0-L_1 and does not increase that on other levels. Third, NoveLSM and RocksDB-L0-NVM have fewer compactions than RocksDB-SSD. The reasons are: (1) NoveLSM uses large mutable MemTables to serve more write requests and absorb a portion of update requests, and (2) RocksDB-L0-NVM has an 8 GB L_0 in NVM to store more data. Fourth, the substantial amount of compaction data in NoveLSM and RocksDB stems from the L_0-L_1 compaction.

Reducing LSM-tree depth: To evaluate the technique of flattening LSM-trees, we change level sizes for both RocksDB



	256MB	8GB	256MB	8GB	256MB	8GB
	RocksDB-SSD		MatrixKV		RocksDB-L0-NVM	
L0	0.00	0.00	0.00	0.00	0.00	0.00
L1	0.19	7.95	0.23	8.01	0.24	7.99
L2	2.47	43.92	2.45	44.67	2.45	41.04
L3	24.97		24.99		25.00	
L4	23.72		21.90		21.85	

Figure 16: Reducing LSM-tree depth. *The y-axis shows random write throughputs of RocksDB and MatrixKV when L_1 is 256 MB/8 GB. The table below shows the data distribution among levels (in GB).*

and MatrixKV. The first configuration is $L_1 = 256MB$ (the default L_1 size of RocksDB). The second configuration is $L_1 = 8GB$. The following levels exponentially increased at the ratio of AF=10. Figure 16 shows the throughput of randomly writing an 80 GB dataset. The table under the figure shows the data distribution on different levels after balancing LSM-trees. The test results demonstrate that both RocksDB and MatrixKV reduce the number of levels by enlarging level sizes, i.e., from 5 to 3. However, they exert opposite influences on system performance. RocksDB-SSD and RocksDB-L0-NVM reduce their random write throughputs by 3× and 1.5× respectively as level sizes increase. The reason is that the enlarged L_1 significantly increases the amount of compaction data between L_0 and L_1. RocksDB-L0-NVM is slightly better than RocksDB-SSD since it puts L_0 in NVMs. For MatrixKV, the throughput increases 25% since the fine granularity column compaction is independent of level sizes. Furthermore, the MatrixKV with 256 MB L_1 shows the performance improvement of only addressing write stalls.

Cross-row hint search: To evaluate the technique of cross-row hint search, we first randomly write an 8 GB dataset with 4 KB value size to fill the L_0 in NVMs for MatrixKV and RocksDB-L0-NVM. Then we search for one million KV items from NVMs in uniformly random order. This experiment makes NVMs accommodate 100% of the dataset to fully reflect the efficiency of cross-row hint searches. The random read throughput of RocksDB-L0-NVM and MatrixKV are 9 MB/s and 157.9 MB/s respectively. Hence, compared to simply placing L_0 in NVMs, the cross-row hint search improves the read efficiency by 17.5 times.

5.4 Extended Comparisons on NVMs

To further verify that MatrixKV's benefits are not solely due to the use of fast NVMs, we evaluate more KV stores on the DRAM-NVM hierarchy, i.e., RocksDB, NoveLSM, PebblesDB, SILK, and MatrixKV, where DRAM stores MemTables, and all other components are stored on NVMs.

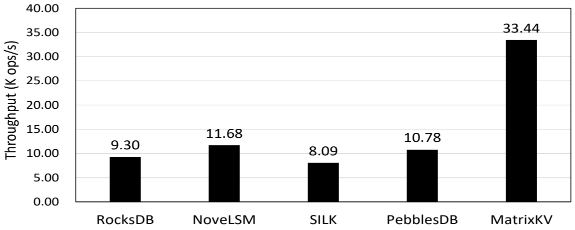


Figure 17: Throughput on NVM based KV stores.

Table 3: Tail latency on NVM-based KV stores

Latency (us)	avg.	90%	99%	99.9%
RocksDB	385	523	701	864
NoveLSM	377	250	808	917
SILK	351	445	575	747
PebblesDB	335	1103	1406	1643
MatrixKV	**209**	**310**	**412**	**547**

Throughput: Figure 17 shows the performance for randomly writing an 80 GB dataset. MatrixKV achieves the best performance among all KV stores. It demonstrates that the enabling techniques of MatrixKV are appropriate for NVM devices. Using NVM as a fast block device, PebblesDB does not show much improvement over RocksDB. SILK is slightly worse than RocksDB since its design strategies have limited advantages over intensive writes.

Tail latency: Tail latencies are evaluated with YCSB-A workload as in § 5.2. Since NVM has a significantly better performance than SSDs, we speed up the requests from clients (60K requests/s). Test results in Table 3 show that with the persistent storage of NVMs most KV stores provide adequate tail latencies. However, MatrixKV still achieves the shortest tail latency.

6 Conclusion

In this paper, we present MatrixKV, a stable low-latency key-value store based on LSM-trees. MatrixKV is designed for systems with multi-tier DRAM-NVM-SSD storage. By lifting the L_0 to NVM, managing it with the matrix container, and compacting L_0 and L_1 with the fine granularity column compaction, MatrixKV reduces write stalls. By flattening the LSM-trees, MatrixKV mitigates write amplification. MatrixKV also guarantees adequate read performance with cross-row hint searches. MatrixKV is implemented on a real system based on RocksDB. Evaluation results demonstrate that MatrixKV significantly reduces write stalls and achieves much better system performance than RocksDB and NoveLSM.

7 Acknowledgement

We thank our shepherd Patrick Stuedi, the anonymous reviewers, and Damien Le Moal for their insightful comments and guidance. We appreciate Yun Liu, Hua Jiang, and Tao Zhong from Intel for the hardware support and maintenance. We also thank Zhiwen Liu and Chenghao Zhu for their efforts in the open-source project.This work was sponsored in part by the National Key Research and Development Program of China No.2018YFB10033005, the Creative Reaserch Group Project of NSFC No.61821003, and the National Natural Science Foundation of China under Grant No.61472152. This work is also partially supported by the US NSF under Grant No.CCF-1704504 and No.CCF-1629625.

References

[1] Persistent memory development kit, 2019. https://github.com/pmem/pmdk.

[2] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP 09)*, pages 1–14, 2009.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*, volume 151. Arpaci-Dusseau Books Wisconsin, 2014.

[4] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758. ACM, 2017.

[5] Anurag Awasthi, Avani Nandini, Arnab Bhattacharya, and Priya Sehgal. Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase. In *Proceedings of the 18th International Conference on Management of Data*, pages 68–79, 2012.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (ATC 19)*, 2019.

[7] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 80–94, 2017.

[8] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[9] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for

storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[10] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010.

[11] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[12] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 11)*, pages 105–118, 2011.

[13] Nachshon Cohen, David T Aksun, Hillel Avni, and James R Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*, pages 441–454, 2019.

[14] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC 10)*, 2010.

[16] George Copeland, Tom W Keller, Ravi Krishnamurthy, and Marc G Smith. The case for safe ram. In *VLDB*, pages 327–335, 1989.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 2017.

[18] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[19] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, pages 449–466, 2019.

[20] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011.

[21] Alexander Driskill-Smith. Latest advances and future prospects of stt-ram. In *Non-Volatile Memories Workshop*, pages 11–13, 2010.

[22] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[23] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, page 42. ACM, 2018.

[24] Facebook. Rocksdb, a persistent key-value store for fast storage enviroments, 2019. http://rocksdb.org/.

[25] Sanjay Ghemawat and Jeff Dean. Leveldb, 2016. https://github.com/google/leveldb.

[26] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[27] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[28] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[29] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, volume 97, pages 16–25, 1997.

[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (ATC 18)*, 2018.

[32] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Phase change memory in enterprise storage systems: silver bullet or snake oil? *ACM SIGOPS Operating Systems Review*, 48(1):82–89, 2014.

[33] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 273–285, 2014.

[34] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16)*, pages 399–411, 2016.

[35] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[36] Lu Lanyue, Pillai Thanumalayan Sankaranarayana, Arpaci-Dusseau Andrea C, and Arpaci-Dusseau Remzi H. WiscKey: separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[37] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[38] Jianhong Li, Andrew Pavlo, and Siying Dong. Nvm-rocks: Rocksdb on non-volatile memory systems, 2017.

[39] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 USENIXAnnual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[40] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 149–166, 2016.

[41] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 1–13, 2011.

[42] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 17)*, pages 329–343, 2017.

[43] Chen Luo and Michael J Carey. On performance stability in lsm-based storage systems (extended version). *arXiv preprint arXiv:1906.09667*, 2019.

[44] Balmau Oana Maria, Didona Diego, Guerraoui Rachid, Zwaenepoel Willy, Yuan Huapeng, Arora Aashray, Gupta Karan, and Konka Pavan. Triad: creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (ATC 17)*, 2017.

[45] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (ATC 15)*, 2015.

[46] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 477–489, 2018.

[47] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. Optimizing key-value stores for hybrid storage architectures. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 355–358, 2014.

[48] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013.

[49] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[50] Patrick E O'Neil and Gerhard Weikum. A log-structured history data access method (lham). In *HPTS*, page 0. Citeseer, 1993.

[51] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[52] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[53] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 12)*, 2012.

[54] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, 2013.

[55] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80, 2008.

[56] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79, 2017.

[57] Doug Terry. *Transactions and Scalability in Cloud Databases—Can't We Have Both?* USENIX Association, Boston, MA, 2019.

[58] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 94)*, pages 86–97, 1994.

[59] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key- value store for small data. In *Proceedings of the USENIX Annual Technical Conference (ATC 15)*, 2015.

[60] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*, pages 427–439, 2019.

[61] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[62] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017)*, 2017.

[63] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 159–171, 2019.

[64] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2016.

[65] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores

Shin-Yeh Tsai, Yizhou Shan, Yiying Zhang
Purdue University and University of California, San Diego

Abstract

Many datacenters and clouds manage storage systems separately from computing services for better manageability and resource utilization. These existing disaggregated storage systems use hard disks or SSDs as storage media. Recently, the technology of persistent memory (PM) has matured and seen initial adoption in several datacenters. Disaggregating PM could enjoy the same benefits of traditional disaggregated storage systems, but it requires new designs because of its memory-like performance and byte addressability.

In this paper, we explore the design of disaggregating PM and managing them remotely from compute servers, a model we call *passive disaggregated persistent memory*, or *pDPM*. Compared to the alternative of managing PM at storage servers, pDPM significantly lowers monetary and energy costs and avoids scalability bottlenecks at storage servers.

We built three key-value store systems using the pDPM model. The first one lets all compute nodes directly access and manage storage nodes. The second uses a central coordinator to orchestrate the communication between compute and storage nodes. These two systems have various performance and scalability limitations. To solve these problems, we built Clover, a pDPM system that separates the location, communication mechanism, and management strategy of the data plane and the metadata/control plane. Compute nodes access storage nodes directly for data operations, while one or few global metadata servers handle all metadata/control operations. From our extensive evaluation of the three pDPM systems, we found Clover to be the best-performing pDPM system. Its performance under common datacenter workloads is similar to non-pDPM remote in-memory key-value store, while reducing CapEx and OpEx by 1.4× and 3.9×.

1 Introduction

Separating (or "disaggregating") storage and compute has become a common practice in many datacenters [11, 18] and clouds [3, 4]. Disaggregation makes it easy to manage and scale both the storage and the compute pools. By allowing the storage pool to be shared across applications and users, disaggregation consolidates storage resources and reduces their cost. As a recent success story, Alibaba listed their RDMA-based disaggregated storage system as one of the five reasons that enabled them to serve the peak load of 544,000 orders per second on the 2019 Single's Day [62].

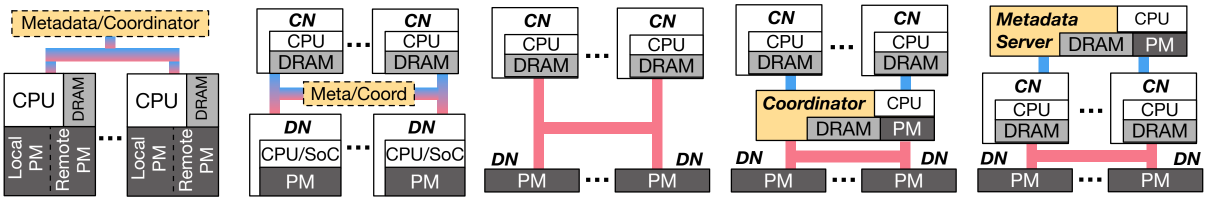
Existing disaggregated storage systems are all SSD- or HDD-based. Today, a new storage media, non-volatile memory (or persistent memory, *PM*) has arrived [27, 29] and has already seen adoption in several datacenters [21, 25, 46]. Existing distributed PM systems [42, 56, 69] have mainly taken a non-disaggregated approach, where each server in a cluster hosts PM for applications running both on the local server and remote servers (Figure 1(a)).

Disaggregating PM could enjoy the same management and resource-utilization benefits as traditional disaggregated storage systems. However, building a PM-based disaggregated system is very different from traditional disaggregated storage systems as PM is byte addressable and orders of magnitude faster than SSDs and HDDs. It is also different from disaggregated memory systems [41, 55], since when treated as storage systems, disaggregated PM systems need to sustain power failure and be crash consistent.

There are two possible design directions in building disaggregated PM systems, and they differ in where management software runs. The first type, and the type that has been adopted in traditional disaggregated storage systems, runs management software at the storage nodes, *i.e.*, actively managing data at where the data is. When applying this model to PM, we call the resulting system *active disaggregated PM*, or *aDPM* (Figure 1(b)). By co-locating data and their management, aDPM could offer low-latency performance to applications. However, aDPM requires significant processing power at storage nodes to sustain high-bandwidth networks and to fully deliver PM's superior performance.

In this paper, we explore an alternative approach of building disaggregated PM by treating storage nodes as *passive* parties that do not perform any data processing or data management tasks, a model we call *pDPM*. pDPM offers several practical benefits and research value. First, pDPM lowers owning and energy cost. Without any processing need, a PM node (we call it a *data node* or *DN*) can either be a regular server that dedicates its entire CPU to other applications or a hardware device that directly attaches a NIC to PM. Second, pDPM avoids DN's processing power being the performance scalability bottleneck. Finally, pDPM is an approach in the design space of disaggregated storage systems that has largely been overlooked in the past. Exploring pDPM systems would reveal various performance, scalability, and cost tradeoffs that could help future researchers and systems builders make better design decisions.

pDPM presents several new challenges, the biggest of which is the need to avoid processing all together from where data is hosted. Existing in-memory data stores heavily rely



(a) non-Disaggregated PM (b) aDPM (c) pDPM-Direct (d) pDPM-Central (e) Clover

Figure 1: **PM Organization Comparison.** *Blue bars indicate two-way communication and pink ones indicate one-way communication. Bars with both blue and pink mean support for both. Dashed boxes mean some but not all existing solutions adopt centralized metadata server (or a coordinator).*

on local processing power for both the data path and the control path. Without any processing power, accesses to DNs have to come all from the network, which makes data operations like concurrent writes especially hard. Moreover, DNs cannot perform any management tasks or metadata operations locally, and each DN can fail independently.

A key question in designing pDPM systems is where to perform data and metadata operations when we cannot perform them at DNs. Our first approach is to let client/compute nodes (*CN*s) perform all the tasks by directly accessing DNs with *one-sided* network communication, a model we call *pDPM-Direct* (Figure 1(c)). After building and evaluating a real pDPM-Direct key-value store system, we found that since CNs cannot be efficiently coordinated, pDPM-Direct performs and scales poorly when there are concurrent reads/writes to the same data. Our second approach is *pDPM-Central* (Figure 1(d)), where we use a central server (the *coordinator*) to manage DNs and to orchestrate all accesses from CNs to DNs. Although pDPM-Central provides a way to coordinate CNs, it adds more hops between CNs and DNs, and the coordinator is a new scalability bottleneck.

To solve the issues of the above two pDPM systems, we build Clover, a key-value store system with a new architecture of pDPM (Figure 1(e)). Clover's main ideas are to separate the location of data and metadata, to use different communication mechanisms to access them, and to adopt different management strategies for them. Data is stored at DNs. Metadata is stored at one or few global metadata servers (*MS*s). CNs directly access DNs for all data operations using *one-sided* network communication. They use *two-sided* communication to talk to MS(s). MS(s) perform all metadata and control operations.

Clover achieves low-latency, high-throughput performance while delivering the consistency and reliability guarantees that are commonly used in traditional distributed storage systems. We designed a set of novel techniques at the data and the metadata plane to achieve these goals. Our data plane design is inspired by log-structured writes and skip lists. This design achieves 1-/2-RTT read/write performance when there is no high write contention, while ensuring proper synchronization and crash consistency of concurrent writes with satisfactory performance. We move all metadata and control operations off performance critical path. We *completely* eliminate the need for the MS to communicate with DNs; it performs space management and other control tasks without accessing DNs. In addition, Clover supports replicated writes for high availability and reliability.

We evaluate Clover, pDPM-Direct, and pDPM-Central using a cluster of servers connected with RDMA network (some acting as CNs and MSs, some acting as emulated DNs). We compare these pDPM systems with two non-disaggregated PM systems [42, 56] and an aDPM key-value store system [30] running on CPU-based servers and on ARM-SoC-based RDMA SmartNIC [44]. We perform an extensive set of experiments to study the latency, throughput, scalability, CPU utilization, and owning cost of these systems using microbenchmarks and YCSB workloads [13, 71]. Our evaluation results demonstrate that Clover is the best-performing pDPM system, and it significantly outperforms traditional distributed PM systems. Clover achieves similar or better performance as aDPM systems under common datacenter workloads, while reducing CapEx and OpEx by 1.4× and 3.9×. However, we also discovered a fundamental limitation of pDPM-based storage systems: no processing at where data sits could hurt write performance, especially under high contention of concurrent accesses to the same data entry. Fortunately, most datacenter workloads are read-most [7]. Thus, we believe pDPM and Clover to be good choices future systems builders can consider, given their overall benefits in cost, performance, and scalability.

Overall, this paper makes the following contributions:

- Thorough exploration of the passive disaggregated persistent-memory architecture, revealing its benefits, tradeoffs, and pitfalls.
- Implementation of Clover and two alternative pDPM key-value stores, all guaranteeing proper synchronization, crash consistency, and high availability.
- A detailed design of how to separate the data plane and the metadata plane under the pDPM model.
- Comprehensive evaluation results that can guide future DPM research.

All our pDPM systems are publicly available at https://github.com/WukLab/pDPM.

2 Background and Related Work

This section includes background and related work on in-memory data stores, RDMA, and PM in datacenter settings.

2.1 PM and Distributed PM Storage

Non-volatile memory (or PM) technologies such as 3D-XPoint [28], PCM, STTM, and the memristor provide byte addressability, persistence, and latency that is within an order of magnitude of DRAM [59, 70]. PM has attracted extensive research efforts in the past decade, most of which focus on single-node environments. The first commercial PM product, Intel Optane DC, has finally come to market [27]. It is pressing to seek solutions to incorporate PM in datacenters.

Existing distributed PM systems [42, 56, 69] have mainly adopted a *symmetric* architecture where each node in a cluster hosts some PM that can be accessed both locally and by other nodes (Figure 1(a)). Some of these systems expose a file system interface [42, 69], and others expose a memory interface [56, 73]. Among them, Orion [69] uses a global server for metadata, and the rest co-locate metadata with data. These systems have fast local-data accesses but lack flexibility in managing compute and storage resources, and they cannot scale these resources independently.

2.2 RDMA and RDMA-Based Data Stores

Remote Direct Memory Access, or *RDMA*, is a network technology that offers low-latency and low-CPU-utilization accesses to memory at remote machines. RDMA supports two communication patterns: *one-sided* and *two-sided*. One-sided RDMA operations allow one node to directly access the memory at another node without involving the latter's CPU. Two-sided RDMA involves both sender's and receiver's CPUs, similar to traditional network messaging.

Because of its performance and cost benefits [22, 36, 49], RDMA has been deployed in major datacenters like Microsoft [63] and Alibaba [2]. Several recent distributed systems such as in-memory key-value stores [15, 16, 38, 47, 48, 58] and in-memory databases/transactional systems [8, 10, 66, 72] use RDMA to perform their network communication. Most of them use a combination of one- and two-sided RDMA or pure two-sided RDMA. For example, FaSST [32] is an RDMA-based RPC system built entirely with two-sided RDMA. FaRM [15, 16], an RDMA-based distributed memory platform, uses one-sided communication for reads and performs both one- and two-sided operations for replicated writes. Pilaf [47] is a key-value store system that uses one-sided RDMA *read* for *get* and two-sided RDMA for *put*. HERD [30, 31] is another RDMA-based key-value store system. For each *get* and *put*, HERD uses two RDMA operations: client sending a one-sided RDMA *write* request to server and server sending an RDMA *send* response to client.

To achieve low-latency performance, most existing systems use busy-polling threads to receive incoming two-sided RDMA requests. They also perform management tasks such as memory allocation and garbage collection at CPUs in data-hosting nodes [15, 72]. Consequently, even when one-sided RDMA operations help reduce CPU utilization, practical RDMA-based data stores still require a CPU and significant amount of energy at each data-hosting server. For example, although FaRM [15] tries to use as much one-sided communication as possible, it still requires processing power at data nodes to perform metadata operations and certain steps in its write replication protocol.

HyperLoop [35] is a recent system that provides a mechanism to extend default one-sided RDMA operations to support more functionalities. These additional functionalities are performed at RDMA NICs without involving host CPU. HyperLoop's computation offloading technique could be applied to pDPM systems to offload certain data operations to DNs, which could potentially improve pDPM's performance. However, it is difficult to offload the more complex metadata operations to RDMA NICs, and HyperLoop still performs them at CPUs. Clover demonstrates how to efficiently separate the metadata plane and run it at a global metadata server.

2.3 Resource Disaggregation

Resource disaggregation is a notion to separate different types of resources into pools (*e.g.*, a compute pool and a storage pool), each of which can be independently managed, configured, and scaled [6, 26, 55]. Because of its efficiency in resource utilization and management, many datacenters and clouds have taken this approach when building storage systems [3, 4, 11, 18, 65].

Disaggregation could take two forms: disaggregating resources and managing them at where they are (active), and disaggregating resources but managing them at the compute pool (passive). Existing storage and memory systems have mainly taken the active approach, with most of them building disaggregated resource pools using regular CPU-based servers [51, 65]. To sustain high-bandwidth networks and fast PM, these systems will require many CPU cores to just poll and process requests. Another way to build active disaggregated systems is to offload computation at storage nodes to hardware [9, 12, 17, 33, 35, 38, 39, 54, 57, 57]. These solutions either require significant hardware implementation efforts (*e.g.*, FPGA-based) or incur performance scalability issues (*e.g.*, ARM-SoC-based).

Compared to active disaggregation, the passive approach of disaggregation largely reduces the owning, energy, and development costs of storage nodes by avoiding busy polling at storage nodes and shifting the rest of the computation to compute nodes. Unfortunately, the passive approach has largely been overlooked in the community. HPE's "The Machine" (Memory-Driven Computing) project [19, 23, 24, 37, 64] is one of the few existing proposals [40, 41] that adopt the passive model. So far, HPE has (separately) built a hardware prototype and a software layer. The hardware prototype [20] connects a set of SoCs to a set of DRAM/PM chips in a rack over a proprietary photonic network. To use this hardware prototype, application developers need to build software layers to manage and access data in DRAM/PM. HPE has also been building a software memory-store solution on top of a Superdome NUMA machine [37, 64]. This solution assumes certain features from future interconnect technologies, does not support data redundancy, and is work done in parallel with us. Although being a significant initial step in passive disaggregation research, the Machine project only explores one design choice and relies heavily on special network to access and manage disaggregated memory. Moreover, its design details are not open to the public.

3 pDPM Overview

This section gives an overview of pDPM, its unique challenges, the interface and guarantees our proposed pDPM systems have, and the network layer they employ. Table 1 summarizes the comparison of our proposed pDPM systems and traditional distributed PM and remote memory systems.

3.1 Passive Disaggregated Persistent Memory

Our definition of the pDPM architecture consists of two concepts: separating PM from compute servers into a PM-based storage pool and eliminating processing needs at these separated PM nodes (DNs).

The first concept is in the same spirit of current disaggregated storage systems and shares many of their benefits: it is flexible to manage and customize the PM storage pool; it offers high resource utilization, since data can be allocated at any DNs; datacenters can scale DNs independently from other servers; and it is easy to add, remove, and upgrade DNs without the need to change existing (compute) servers.

The second concept follows the more aggressive disaggregation approach of forming resource pools with just hardware (PM in our case). Such PM pools can be a set of regular servers equipped with PM or a set of network-attached devices with just network functionality and some PM. The former frees entire server CPUs to perform other tasks, while the latter eliminates the need for a processor and its hardware/server packaging all together, reducing not only the energy cost but also the building cost of DNs. Moreover, by removing processing from DNs, pDPM also avoids DN-side processor being a performance scalability bottleneck.

3.2 pDPM Challenges

pDPM offers many cost and manageability benefits and is now feasible to build with fast, “one-sided” network communication technologies like RDMA. However, it is only attractive when there is no or minimal performance loss compared to other more expensive solutions. Building a pDPM storage system that can lower the cost but maintain the performance of non-pDPM systems is hard. Different from traditional distributed storage and memory systems, DNs can only be accessed *and* managed remotely. A major technical hurdle is in providing good performance with concurrent data accesses. The lack of processing power at DNs makes it impossible to orchestrate (e.g., serialize) concurrent accesses there. Managing distributed PM resources without any pDPM-local processing is also hard and when performed improperly, can largely hurt foreground performance. In addition, DNs can fail independently. Such failures should be handled properly to ensure data reliability and availability.

Different from traditional disaggregated storage that is based on SSDs or hard disks, PM is orders of magnitude faster [70]. Although today’s datacenter network speed has also improved significantly [43], pDPM storage systems should still try to minimize network RTTs.

Different from disaggregated memory systems [40, 41, 55], pDPM is a persistent storage system and should sustain power failures and node failures. Thus, we need to ensure the consistency of data and metadata during crash recovery and provide redundancy for high availability and reliability.

3.3 System Interface and Guarantees

Clover and our two alternative pDPM systems provide the same interface and guarantees to applications. They are key-value stores supporting variable-sized entries, where users can create, read (get), write (put), and delete a key-value entry. Different CNs can have shared access to the same data. All our pDPM systems ensure the atomicity of an entry across concurrent readers and writers. A successful write indicates that the data entry is committed (atomically). Reads only see committed value. We choose to build key-value stores on the pDPM architectures because key-value stores are widely used in many datacenters. Similarly, we choose single-entry atomic write and read committed because these consistency and isolation levels are widely used in many data store systems [30, 47] and can be extended to other levels.

Our pDPM systems are intended for storing data persistently. They provide crash consistency, data reliability, and high availability. After recovering from crashes at arbitrary points, each data entry is guaranteed to contain either only new data values or only old ones. In addition, all our three systems support replicated writes across DNs.

3.4 Network Layer

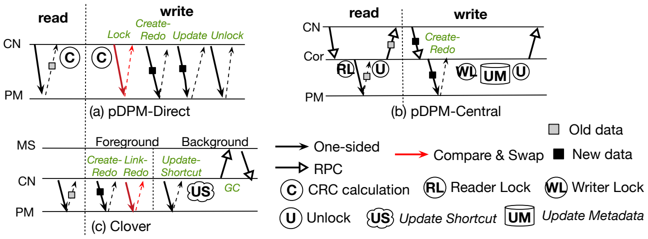
We choose RDMA as the network technology to connect all servers and DNs. We use RDMA’s RC (Reliable Connection) mode which supports one-sided RDMA operations and ensures lossless and ordered packet delivery. Similar to prior solutions [15, 30, 31, 67], we solve RDMA’s scalability issues by registering memory regions using huge memory pages with RDMA NICs. Note that we use regular RDMA writes as persistent write for our evaluation, since the RDMA durable write commit in the IETF standard takes one network round trip [60], same as non-durable RDMA write.

4 Alternative pDPM Systems

Before Clover, we built two other pDPM systems during our exploration of the pDPM architectures. They follow the same interface and deliver the same consistency and relia-

System	CapEx ($)	R-RTT	W-RTT	Energy	Scalability	Metadata	Performance
Distributed PM	46736	0-N	0-N	High	Neither	Large	Good only when accessing data on local node
aDPM w/ CPU	79888	1	1	High	w/ both⋆	Small	Good overall
aDPM w/ BlueField	80080	1	1	Low	Neither	Small	Good under light load
pDPM-Direct	53096	1	4(4)	Low	w/ DN†	Large	Best for small-sized read
pDPM-Central	58096	2	2(2)	High	Neither	Small	Not good for read-intensive traffic
Clover	58096	1	2(3)	Low	w/ both	Medium	Good overall (unless high write contention)

Table 1: **Comparison of Distributed PM Architectures.** *The CapEx column represents dollar cost to build eight CNs and eight PMs. Section 6 discusses the details of CapEx and energy (CPU utilization) calculation. The R-RTT and W-RTT columns show the number of RTTs required to perform a read and a write (with replication). All RTT values are measured when there is no contention. RTTs in distributed PM's read/write, N, depends on protocols and whether data is local. The Scalability column shows if a system is scalable with the number of CNs, the number of DNs, both, or neither. ⋆ only when there are enough CPU cores. † only scalable when there is no contention. The metadata columns show the space needed to store the metadata of a data entry.*



(a) pDPM-Direct

(b) pDPM-Central

(c) Clover

Figure 2: **Read/Write Protocols of pDPM Systems.**

bility guarantees as Clover. Even though the main system we present in this paper is Clover, we have spent significant amount of efforts on optimizing the performance of these alternative systems and on adding replication and crash recovery support to them. They can be used as stand-alone systems apart from being comparison points of Clover. Because of space constraint, we only briefly present their basic data structures and read/write protocols. We omit the discussion of their replication and crash recovery protocols.

4.1 Direct Connection

Our first alternative pDPM system, *pDPM-Direct*, connects CNs directly to DNs (Figure 1(c)). CNs perform unorchestrated, direct accesses to DNs using one-sided RDMA operations. The main challenge in designing pDPM-Direct is the difficulty in coordinating CNs for various data and metadata operations.

To avoid frequent space allocation (which requires coordination across CNs), we pre-assign two spaces for each data entry, one to store committed data where reads go to (the *committed space*) and one to store in-flight, new data (the *uncommitted space*). CNs allocate these spaces at data-entry creation time with the help of a distributed consensus protocol. Afterwards, their locations do not change until data-entry free time. CNs locally store all the metadata (*e.g.*, the locations of committed and uncommitted spaces) to avoid reading and writing metadata to DNs and the cost of ensuring metadata consistency under concurrent accesses.

To support synchronized concurrent data accesses and to avoid reading intermediate data during concurrent writes, a straightforward method and our strawman version is to always lock a data entry when reading or writing it using a distributed lock. Doing so incurs two additional network RTTs for each data access (one for lock and one for unlock).

For better performance, we adopt a lock-free, checksum-based read mechanism, which allows reads to take only one RTT. Specifically, we associate a CRC (error detection code) checksum with each key-value entry at DNs. To read a data entry, a CN uses its stored metadata to find the location of the data entry's committed space. It then reads both the data and its CRC from the DN with an RDMA read. Afterwards, the CN calculates the CRC of the fetched data and compares this calculated CRC with the fetched CRC. If they do not match, then the read is incomplete (an intermediate state during an ongoing write), and the CN retries the read request. Although calculating CRCs adds some performance overhead, it is much lower than the alternative of locking. Figure 2(a) illustrates pDPM-Direct's read and write protocols.

pDPM-Direct still requires locking for writes. We designed an efficient, RDMA-based implementation of write lock. We associate an 8-byte value at the beginning of each data entry as its lock value. To acquire the lock, a CN performs an RDMA `c&s` (compare-and-swap) operation to the value. The `c&s` operation compares whether the value is 0. If so, it sets it to 1. Otherwise, the CN retries the `c&s` operation. To release the lock, the CN performs an RDMA write and sets the value to 0. Our lock implementation leverages the unique feature of the pDPM model that all memory accesses to DNs come from the network (i.e., the NIC). Without any yprocessor's accesses to memory, the DMA protocol guarantees that network atomic operations like `c&s` are atomic across the entire DN [14, 61].

To write a data entry, a CN first calculates and attaches a CRC to the new data entry. Afterwards, the CN locates the entry with its local metadata and locks the entry (one RTT). The CN then writes the new data (with the CRC) to the un-committed space (one RTT), which serves as the *redo* copy used during recovery if a crash happens. Afterwards, the CN writes the new data to the committed space with an RDMA write (one RTT). At the end, the CN releases the lock (one RTT). The total write latency is four RTTs plus the CRC calculation time (when no contention), and two of these RTTs contain data.

4.2 Connecting Through Coordinator

Our second alternative pDPM system, *pDPM-Central* (Figure 1(c)), uses a central *coordinator* to orchestrate all data accesses and to perform metadata and management operations. All CNs send RPC requests to the coordinator, which handles them by performing one-sided RDMA operations to DNs. We implement our RPC using HERD's RPC design [30]; other RPC designs can easily be integrated too. To achieve high throughput, we use multiple RPC handling threads at the coordinator. Figure 2(b) illustrates pDPM-Central's read and write protocols.

Since all requests go through the coordinator, it can serve as the serialization point for concurrent accesses to a data entry. We use a local read/write lock for each data entry at the coordinator to synchronize across multiple coordinator threads. In addition to orchestrating data accesses, the coordinator performs all space allocation and de-allocation of data entries. The coordinator uses its local PM to persistently store all the metadata of a data entry.

To perform a read, a CN sends an RPC read request to the coordinator. The coordinator finds the location of the entry's committed data using its local metadata, acquires its local lock of the entry, reads the data from the DN using a one-sided RDMA read, releases its local lock, and finally replies to the CN's RPC request. The end-to-end read latency a CN observes (when there is no contention) is two RTTs, and both RTTs involve sending data.

To perform a write, the coordinator allocates a new space at a DN for the new data and then writes the data there. We do not need to lock (either at coordinator or at the DN) during this write, since it is an out-of-place write to a location that is not exposed to any other coordinator RPC handlers. After the write, the coordinator updates its local metadata with the new data's location and flushes this new location to its local PM for crash resistance. The total write latency without contention is two RTTs, both containing data.

4.3 pDPM-Direct/-Central Drawbacks

pDPM-Direct delivers great read performance when read size is small, since it only requires one lock-free RTT and it is fast to calculate small CRC. Its write performance is much worse because of high RTTs and lock contention. Its scalability is also limited because of lock contention during concurrent writes. Moreover, pDPM-Direct requires large space for both data and metadata. For each data entry, it doubles the space because of the need to store two copies of data. The metadata overhead is also high, since all CNs have to store all the metadata.

pDPM-Central largely reduces write RTTs over pDPM-Direct and thus has good write performance when the scale of the cluster is small. Unlike pDPM-Direct, CNs in pDPM-Central do not need to store any metadata. However, from our experiments, the coordinator soon becomes the performance bottleneck when either the number of CNs or the number of DNs increases. pDPM-Central's read performance is also worse than pDPM-Direct with the extra hop between a CN and the coordinator. In addition, the coordinator's CPU utilization is high, since it needs many RPC handler threads to sustain parallel requests from CNs.

5 Clover

To solve the problems of the first two pDPM systems we built, we propose Clover (Figure 1(e)). The main idea of Clover is to separate the location, the communication method, and the management strategy of the data plane and the control plane. It lets CNs directly access DNs for all data operations and uses one or few metadata servers (*MS*s) for all control plane operations.

To avoid MS being the scalability bottleneck, we support multiple MSs, each serving a shard of data entries. Each MS stores the metadata of the data entries it is in charge of in its local PM. We keep the amount of metadata small. The storage overhead of metadata is below 2% for 1 KB data entries. CNs cache the metadata of hot data entries. Under memory pressure, CNs evict metadata with a replacement policy (we currently support FIFO and LRU).

Clover aims to deliver scalable, low-latency, high-throughput performance at the data plane and to avoid the MS being the bottleneck at the control plane. Our overall approaches to achieve these design goals include: 1) moving all metadata operations off performance critical path, 2) using lock-free data structures to increase scalability, 3) employing optimization mechanisms to reduce network round trips for data accesses, and 4) leveraging the unique atomic data access guarantees of pDPM. Figure 2(c) shows the read and write protocol of Clover. Figure 3 illustrates the data structures used in Clover.

5.1 Data Plane

To achieve our data plane design goals, we propose a new mechanism to perform lock-free, fast, and scalable reads and writes. Specifically, we allow multiple committed versions of a data entry in DNs and link them into a *chain*. Each committed write to a data entry will move its latest version to a new location. To avoid the need of updating CNs with the new location, we use a self-identifying data structure for CNs to find the latest version.

We include a *header* with each version of a data entry. The header contains a *pointer* and some metadata bits used for garbage collection. The pointers chain all versions of a data entry together in the order that they are written. A `NULL` pointer indicates that the version is the latest.

A CN acquires the header of the chain head from the MS at the first access to a data entry. It then caches the header locally to avoid the overhead of contacting MS on every data access. We call a CN-cached header a *cursor*.

Read. Clover reads are lock-free. To read a data entry, a CN performs a *chain walk*. The chain walk begins with fetching

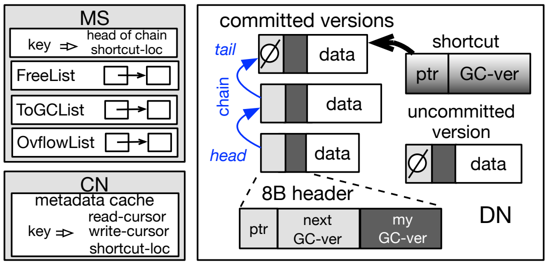


Figure 3: **Clover System Design.**

the data buffer version that the CN's cursor points to. It then uses the pointer in this fetched buffer to read the next version. The CN repeats this step until reading a `NULL` pointer, which indicates that it has read the latest version. All steps in the chain walk use one-sided RDMA reads. After a chain walk, the CN updates its cursor to point to the latest version.

A chain walk can be slow with long chains when a cursor is not up to date [68]. Inspired by skip lists [53], we solve this issue by using a *shortcut* to directly point to the latest version or a recent version of each data entry. Shortcuts are *best effort* in that they are intended but not enforced to always point to the latest version of an entry. The shortcut of a data entry is stored at its DN. The location of a shortcut never changes during the lifetime of the entry. MS stores the locations of all shortcuts. When a CN first accesses a data entry, it retrieves the location of its shortcut from MS and caches it locally.

The CN issues a chain walk read and a shortcut read in parallel. It returns the user request when the faster one finishes and discards the other result. We do not replace chain walks completely with shortcut reads, since shortcuts are updated asynchronously in the background and may not be updated as fast as the cursor. When the CN's cursor points to the latest version of a data entry, a read only takes one RTT.

Write. Clover never overwrites existing data entries and performs a lock-free, out-of-place write before linking the new data to an entry's chain. To write a data entry, a CN first selects a free DN space assigned to it by MS in advance (see §5.2). It performs a one-sided RDMA write to write the new data to this buffer. Afterwards, the CN performs an RDMA `c&s` operation to link this new data to the tail of the entry's version chain. Specifically, the `c&s` operation is on the header that the CN's cursor points to. If the pointer in the header is `NULL`, the `c&s` operation swaps the pointer to point to the new data, and we treat this new data as a *committed* version. Otherwise, it means that the cursor does not point to the tail of the chain and the CN performs a chain walk to reach the tail and then issues another `c&s`.

Afterwards, the CN uses a one-sided RDMA write to update the shortcut of the entry to point to the new data version. This step is off the performance critical path. The CN also updates its cursor to the newly written version. We do not invalidate or update other CNs' cursors at this time to improve the scalability and performance of Clover.

Clover' chained structure and write mechanism ensure that writers do not block readers and readers do not block writers. They also ensure that readers can only view committed data. Without high write contention to the same data entry, one write takes only two RTTs.

Retire. After committing a write, a CN can *retire* older versions of the data entry, indicating that the buffer spaces can be reclaimed. To improve performance and minimize the need to communicate with MS, CNs lazily send asynchronous, batched retirement requests to MS in the background. We further avoid the need for MS to invalidate CN-cached metadata using a combination of timeout and epoch-based garbage collection (see §5.2).

5.2 Control Plane

CNs communicate with MS using two-sided operations for all metadata operations. MS performs all types of management of DNs. We carefully designed the MS functionalities for best performance and scalability.

Space allocation. With Clover's out-of-place write model, Clover has high demand for DN space allocation. We use an efficient space allocation mechanism where MS packages free spaces of all DNs into *chunks*. Each chunk hosts data buffers of the same size. Different chunks can have different data sizes. Instead of asking for a new free space before every write, each CN requests multiple spaces at a time from MS in the background. This approach moves space allocation off the performance critical path and is important to deliver good write performance.

Garbage collection. Clover' append-only chained data structure makes its writes very fast. But like all other append-only and log-structured storage systems, Clover needs to garbage collect (GC) old data. We design a new efficient GC mechanism that does not involve *any* data movement or communication to DN. It also minimizes the communication between MS and CNs.

The basic flow of GC (a strawman implementation) is simple: MS processes incoming retire requests from CNs by putting reclaimed spaces to a free list (*FreeList*). It gets free spaces from the FreeList when a CN requests more free buffers. A free space can be used by any CN for any new writes, as long as the size fits.

Although the above strawman implementation is simple, making GC work correctly, efficiently, and scale well is challenging. First, to achieve good GC performance, we should avoid the invalidations of CN cached cursors after reclaiming buffers to minimize the network traffic between MS and CNs. However, with the strawman GC implementation, CNs' outdated cursors can cause failed chain walks. We solve this problem using two techniques: 1) MS does not clear the header (or the content) of a data buffer after reclaiming it, and 2) we assign a *GC version* to each data buffer. MS increases the GC version after reclaiming a data buffer. It gives this new GC version together with the location of the buffer when assigning the buffer as a free space to a CN, CN_k. Before CN_k uses the space for a new write, the content of this space at the DN contains old data and old GC version. After CN_k uses the space for a write, it contains new data and new GC version. Other CNs that have cached cursors to this buffer need to differentiate these two cases. A CN tells if a buffer contains its intended data by comparing the GC version in its cached cursor to the one it reads from the DN. If they do not match, the CN will discard the read data and invalidate its cached cursor. Our GC-version approach not only avoids the need for MS to invalidate cursor caches on CNs, but also eliminates the need for MS to access DNs during GC.

The next challenge is related to our goal of read-isolation and atomicity guarantees (*i.e.*, readers always read the data that is consistent to its metadata header). An inconsistent read can happen if the read to a data buffer takes long, and during the reading time, this buffer has been retired by another CN, reclaimed by MS, assigned to a CN as a newly allocated buffer, and used to perform a write. We use a read timeout scheme similar to FaRM [15] to prevent this inconsistent case. Specifically, we abort a read operation after two RTTs, since the above steps in the problematic case take at least (and usually a lot more than) two RTTs (one for a CN to submit the retirement request to MS and one for MS to assign the space to a new CN).

The final challenge is the overflow of GC versions. We can only use limited number of bits for GC version in the header of a data buffer (currently 8 bits), since the header needs to be smaller than the size of an atomic RDMA operation. When the GC version of a buffer increases beyond the maximum value, we have to restart it from zero. With just our GC mechanism so far, CNs will have no way to tell if a buffer matches its cached cursor version or has advanced by $2^8 = 256$ versions. To solve this rare issue without invalidation traffic to CNs, we use an epoch-based timeout mechanism. When MS finds the GC version of a data buffer overflows, it puts the reclaimed buffer into an *OvflowList* and waits for T_e (a configurable time value) before moving it to the *FreeList*. All CNs invalidate their own cursors after an inactive period of T_e (if during this time, the CN access the buffer, it would have advanced the cursor already). To synchronize epoch time, MS sends a message to CNs after T_e. Epoch messages are the only communication from MS to CNs during GC.

5.3 Discussion

The Clover design offers four benefits. First, Clover yields the best performance among all pDPM systems; it outperforms pDPM-Direct and pDPM-Central for both reads and writes, and both with and without contention. Achieving this low latency while guaranteeing atomic write and read committed is not easy. Four approaches enable us to reach this goal: 1) ensuring that the data path does not involve MS, 2) reducing metadata communication to MS and moving it off performance critical paths, 3) ensuring no memory copy in the whole data path, and 4) leveraging the unique advantages of pDPM to perform RDMA atomic operations.

Second, Clover scales well with the number of CNs and DNs, since its reads and writes are both lock free. Readers do not block writers or other readers and writers do not block readers. Concurrent writers to the same entry only contend for the short period of an RDMA `c&s` operation. Clover also minimizes the network traffic to MS and the processing load on MS, which enables MS to scale well with the number of CNs and with the amount of data operations.

Third, we avoid data movement and communication between MS and DNs entirely during GC. To scale and support many CNs with few MSs, we also avoid CN invalidation messages. MS does not need to proactively send any other messages to CNs either. Essentially, MS never *pushes* any messages to CNs. Rather, CNs *pull* information from MS. Furthermore, MS adopts a thread model that adaptively lets working threads sleep to reduce MS's CPU utilization.

Finally, the Clover data structure is flexible and can support load balancing very well. Different versions of a data entry do not need to be on the same DN. As we will see in §5.4 and §5.5, this flexible placement is the key to Clover's load balancing and data replication needs.

However, Clover also has its limitation. Each write in Clover requires two RTTs and under heavy contention, its write performance degrades. As we will see in §6, two-sided aDPM systems outperform Clover with write-intensive workloads, since they can complete writes in one RTT. Fortunately, most datacenter workloads are read-most [7], and under common cases, Clover delivers great performance.

5.4 Failure Handling

DNs can fail independently from CNs. Clover needs to handle both transient and permanent failures of a DN. For the former, Clover guarantees that a DN can recover all its committed data after reboot (*i.e.*, crash consistent). For the latter, we add the support of data replication across multiple DNs to Clover. In addition, Clover also handles the failure of MS.

Recovery from transient failures. Clover's recovery mechanism of a single DN's transient failure is simple. If a DN fails before a CN successfully links the new data it writes to the chain (indicating an un-committed write), the CN simply discards the new write by treating the space as unused.

Data redundancy. With the user-specified degree of replication being N, Clover guarantees that data is still accessible after $N-1$ DNs have failed. We propose a new atomic replication mechanism designed for the Clover data structure.

The basic idea is to link each version of a data entry D_N to all the replicas of the next version (*e.g.*, D^a_{N+1}, D^b_{N+1}, D^c_{N+1} for three replicas) by placing pointers to all these

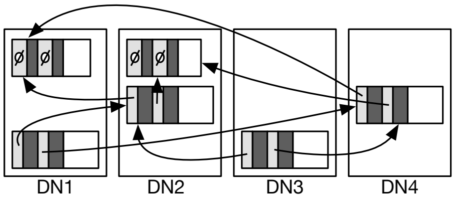


Figure 4: **Replicated Data Entry.** *A replicated data entry on four DNs. The replication factor is two.*

replicas in the header of D_N. Figure 4 shows an example of a replicated data entry (with the degree of replication being 2). With this all-way chaining method, Clover can always construct a valid chain as long as one copy of each version in a data entry survives.

Each data entry version has a primary copy and one or more secondary copies. To write a new version, D_{N+1}, to a data entry whose current tail is D_N with R replicas, a CN first writes the new data to R DNs. In parallel, the CN performs a one-sided `c&s` to a bit, B_w, in the header of the primary copy of D_N to test if the entry is already in the middle of a replicated write. If not, the bit will be set, indicating that the entry is now under replicated write. All the writes and the `c&s` operation are sent out in parallel to minimize latency. After the CN receives the RDMA acknowledgment of all the operations, it constructs a header that contains R pointers to the copies of D_{N+1} and writes it to all the copies of D_N. Once the new header is written to all copies of D_N, the system can recover D_{N+1} from crashes (up to $R-1$ concurrent DN failure).

MS redundancy. MSs manage several types of metadata. Among them, the only type of metadata that cannot be reconstructed is keys (of key-value entries) and the mapping from a key to the location of its data entries in DNs. To avoid MS being the single point of failure, we implement a mechanism to include one or more *backup* MS. When creating (deleting) a new key-value data entry, the primary MS synchronously replicates (removes) the key and the head of the value chain to all the backup MSs. These metadata are the only metadata that cannot be reconstructed. MSs reconstruct all other metadata by reading value chains in DNs.

5.5 Load Balancing

A pDPM system has a pool of DNs. It is important to balance the load to each of them. We use a novel two-level approach to balance loads in Clover: globally at MS and locally at each CN. Our global management leverages two features in Clover: 1) MS is the party that assigns all new spaces to CNs, and 2) data versions of the same entry in Clover can be placed on different DNs. To reduce the load of a DN, MS assigns more free spaces from other DNs to CNs at allocation time. Each CN internally balances the load to different DNs at runtime. Each CN keeps one bucket per DN to store free spaces. It chooses free spaces from different buckets for new writes according to its own load balancing needs.

6 Evaluation Results

This section presents the evaluation results of Clover. We compare it with pDPM-Direct, pDPM-Central, two distributed PM-based systems, Octopus [42] and Hotpot [56], and a two-sided RDMA-based key-value store, HERD [30]. All our experiments were carried out in a cluster of 14 machines, connected with a 100 Gbps Mellanox InfiniBand Switch. Each machine is equipped with two Intel Xeon E5-2620 2.40GHz CPUs, 128 GB DRAM, and one 100 Gbps Mellanox ConnectX-4 NIC.

In order to compare the pDPM architecture with a low-cost aDPM architecture, we use Mellanox BlueField, a SmartNIC that includes an ARM-based SoC and a 100 Gbps Mellanox ConnectX-5 NIC [44]. We port HERD to BlueField by migrating it from x86 to ARM (we call the ported HERD running on BlueField HERD-BF).

Unfortunately, at the time of writing, we cannot get hold of real Intel Optane DC, and we use DRAM as PM. Our experiments run on machines with PCIe 3.0 ×8 (7.87 GB/s), and the bandwidth from RDMA-NIC to DRAM is capped by it, making the effective bandwidth at most 7.87 GB/s. Intel Optane DC's read bandwidth is 6.6 GB/s [70], which is close to PCIe 3.0 ×8. Thus, we envision read results to be similar with real Optane. Optane's write bandwidth is 2.3 GB/s, and there may be some difference in our write results with real Optane. But since our target is read-most workloads, we believe that the conclusion we make from our evaluation will still be valid with real PM.

6.1 Micro-benchmark Performance

We first evaluate the basic read/write latency of Clover and the systems in comparison using a simple micro-benchmark where a CN synchronously reads/writes a key-value data entry on a DN. For this and all the rest of our experiments, we use HERD's default configuration of 12 busy polling receiving side's threads for both HERD and HERD-BF.

Figure 5 plots the read latency with different request sizes. We use native RDMA one-sided read as the baseline. Overall, Clover's performance is the best among all systems and is only slightly worse than native RDMA. pDPM-Direct has great read performance when the request size is small. However, when request size increases, the overhead of CRC calculation dominates, largely hurting pDPM-Direct's read performance. As expected, pDPM-Central's read performance is not good because of its 2-RTT read protocol. HERD performs worse than Clover because it requires some extra CPU processing time for each read. HERD-BF has a constant overhead over HERD mainly because its processing is performed in BlueField's low-power ARM cores.

Figure 6 plots the average write latency comparison. We use native RDMA one-sided write as a baseline here. Among pDPM systems, Clover and pDPM-Central achieve the best write latency. pDPM-Direct's write performance is the worst

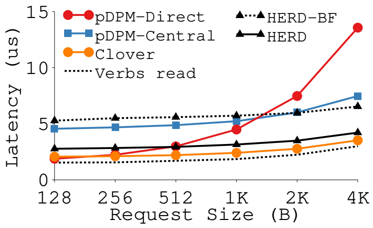


Figure 5: **Read Latency.**

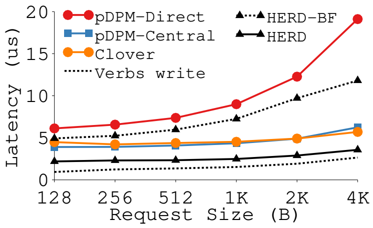


Figure 6: **Write Latency.**

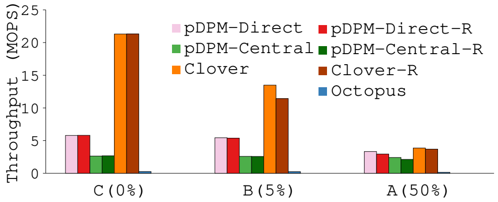


Figure 7: **Throughput Comparison with YCSB.** *Running YCSB on four CNs and four DNs.*

because of its 4-RTT write protocol. Its write performance also gets worse with larger request size because of the increased overhead of CRC calculation. HERD outperforms Clover and other pDPM systems because two-sided communication allows it to complete a write within one RTT. HERD-BF is still a lot worse than HERD because of BlueField's low processing power.

6.2 YCSB Performance and Scalability

We now present our evaluation results using the YCSB benchmark [13, 71]. We use a total of 100K key-value entries and 1M operations per test. The accesses to keys follow the Zipf distribution. We use three workloads with different read-write ratios: read only (workload C), 5% write (workload B), and 50% write (workload A). These three workloads follow common application patterns in datacenters [7] and are the set that previous PM and in-memory store systems used for evaluation [30, 38, 47, 69].

Basic performance. We first evaluate the performance of Clover, pDPM-Direct, pDPM-Central, Octopus, and Hotpot under the same configuration: 4 CNs and 4 DNs, each CN running 8 application threads. Neither Octopus nor Hotpot directly support key-value interface. In order to run the YCSB key-value store workloads, we run MongoDB [50], a key-value store database, on top of Octopus and Hotpot. Note that HERD only supports one DN and we cannot compare with HERD or HERD-BF in this experiment. We also evaluate replication with our three pDPM systems here (with degree of replication 2). Figure 7 shows the overall performance of these systems. Hotpot yields similar performance as Octopus and we omit its results in the figure.

Clover performs the best among all systems for all workloads. We further look into the Clover results and find that the average number of hops during chain walks is only 0.2 to 0.3 for reads and 3.7 to 3.9 for writes. pDPM-Direct performs better with read-most workloads than write-most workloads. This is because without the need to perform any locking, its read performance is not affected by contention. pDPM-Central's performance is the worst among pDPM systems, because under contention (Zipf distribution), the coordinator becomes the bottleneck.

The overall performance of Octopus and Hotpot is more than an order of magnitude worse than all pDPM systems. There are two main reasons. First, these systems do not directly support key-value interface, and running MongoDB on top of them adds overhead. Unfortunately, there is no existing distributed PM systems that directly support key-value interface as far as we know. Second, each read and write operation in these systems involves a complex protocol that requires RPCs across multiple nodes.

Workload	Median	Average	99%
C	1	1	1
B	1	1.26	5
A	1	1.33	6

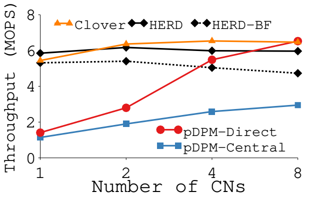
Table 2: **Clover RTTs.**

To further understand Clover's performance, we measure the number of RTTs incurred when running YCSB on Clover. Table 2 shows the median, average, and 99% RTTs of Clover. Clover requires only one RTT for read-most workloads. Even for 50% write (workload A), Clover only incurs six RTTs at 99% and one RTT at median.

Replication overhead. As expected, adding redundancy lowers the throughput of write operations for all pDPM systems. Even though these systems issue the replication requests in parallel, they only use one thread to perform asynchronous RDMA read/write operations, and doing so still has an overhead. However, the overhead is small.

Scalability. Next, we evaluate the scalability of different systems with respect to the number of CNs and the number of DNs. Figure 8 shows the scalability of pDPM systems, HERD, and HERD-BF when varying the number of CNs with a single DN. Clover and HERD have the best (and similar) performance with workload C. Both systems saturate network bandwidth, and neither have any scalability bottlenecks. With workload B, the performance of Clover is slightly worse than HERD because of increased write contention. HERD-BF performs worse and scales worse than Clover and HERD for both workloads mainly because of its limited processing power. pDPM-Central performs the worst and does not scale well with more CNs. pDPM-Direct also performs poorly with fewer CNs. Apart from the limitation of these system's designs, their inefficient thread models also contribute to their worse performance.

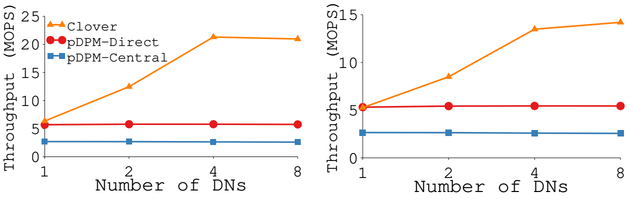
Figure 9 shows the scalability of pDPM data stores w.r.t. the number of DNs (HERD only supports single memory node and we cannot include it in this experiment). Clover scales well with DNs because CNs access DNs directly for data accesses, having no scalability bottleneck. pDPM-



(a) Workload C (0%)

(b) Workload B (5%)

Figure 8: **Scalability w.r.t. CNs.** *Running 1 DNs.*



(a) Workload C (0%)

(b) Workload B (5%)

Figure 9: **Scalability w.r.t. DNs.** *Running 4 CNs.*

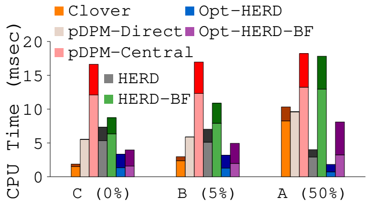


Figure 10: **CPU Utilization.** *Lighter colors and darker colors represent the CPU time used by the client side and the server side. Opt-HERD and Opt-HERD-BF are hypothetical optimal values.*

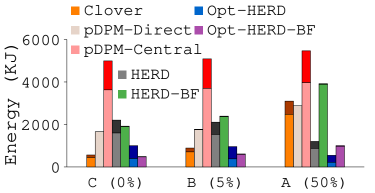


Figure 11: **Energy Consumption.** *Lighter colors and darker colors represent the CPU time used by the client side and the server side. Opt-HERD and Opt-HERD-BF are hypothetical optimal values.*

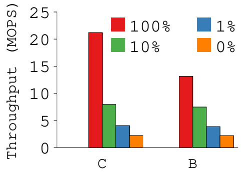


Figure 12: **Effect of Metadata Cache in Clover.**

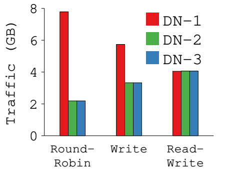


Figure 13: **Load Balancing in Clover.**

Central has poor scalability because of the coordinator being the bottleneck that all requests have to go through. Surprisingly, pDPM-Direct's scalability is also poor. This is because when the number of DNs increases, network bandwidth has not become a performance bottleneck, but CNs need to do more CRC calculation to read/write to more DNs. This computation overhead becomes the performance bottleneck.

6.3 CPU Utilization and Cost

CPU utilization and energy cost. We evaluate the CPU utilization of different systems by calculating the total CPU time to complete ten million requests in YCSB's workloads A, B, and C, as illustrated in Figure 10. We further separate the CPU time used at client side (CNs) and at server side (DNs, the coordinator, MS). The three pDPM systems run four CNs and four DNs. HERD and HERD-BF run four CNs and one DN. Since HERD only supports one DN, to estimate the CPU utilization and energy of a scale-out version of HERD, we hypothetically assume that HERD can achieve perfect scaling (*i.e.*, we reduce HERD's total run time by a factor of four to model it running on four CNs and four DNs). This hypothetical calculation is the optimal performance HERD could have achieved.

We further calculate the total energy cost using the power consumption of our CPU core [34] and the ARM core of BlueField [52]. Figure 11 plots this result. We do not include the energy cost of PM, since it is the same for all systems.

For read-most workload, pDPM-Direct and Clover use less CPU time than pDPM-Central and HERD because they perform one-sided RDMA directly from CNs to DNs. HERD's total CPU time is much longer than Clover even with optimal scale-out calculation, because it uses many busy-polling threads at its server side to achieve good performance (12 threads by default). Surprisingly, HERD-BF's energy is higher than HERD even when the power consumption of an ARM core is more than an order of magnitude lower than our CPU core. HERD-BF's worse performance makes each request to run longer and consumes more power. pDPM-Central has high CPU utilization because the coordinator's CPU spends time on every request, and the total time to finish the workloads with pDPM-Central is long. HERD's write performance and energy are both better than Clover. pDPM systems consume more energy for write-heavy workloads because of their degraded write performance.

CapEx. Table 1 summarizes the cost to build different data stores with 8 CNs and 8 DNs (for distributed PM, we use eight machines in total). The CapEx is calculated with the market price of 128 GB Intel Optane PM ($842 [5]), Mellanox ConnectX-4 NIC ($795 [45]), Mellanox BlueField NIC ($4168 [1]), and a DELL R740 server with the same configuration as what we use in our experiments ($5000). For servers with PM, we adjust the price difference between PM and DRAM to the whole server price ($4144). Distributed PM has the lowest CapEx because it can share PM and only needs eight machines in total. aDPM with CPU requires 16 machines in total (8 for CNs and 8 for DNs). The three pDPM systems and aDPM with BlueField do not require full machines for DNs and we only include PM and NIC costs for them. Surprisingly, the cost of BlueField is similar to a full machine. We suspect that this is because BlueField is in a new and small market, and we expect its price to drop in the future (but still not as low as pDPM).

6.4 Metadata Caching

Each data entry in Clover requires a constant of 8 B metadata (which is much smaller than typical key-value sizes of 100 B - 1000 B [7]). To evaluate the effect of different sizes of metadata cache at CNs in Clover, we ran the same YCSB workloads and configuration as Figure 7 and plot the results in Figure 12. Here, we use the FIFO eviction policy (we also tested LRU and found it to be similar or worse than FIFO). With smaller metadata cache, all workloads' performance drop because a CN has to get the metadata from the MS before accessing a data entry that is not in the local metadata cache. With no metadata cache (0%), CNs need to get metadata from the MS before every request. However, under Zipf distribution, with just 10% metadata cache, Clover can already achieve satisfying performance.

6.5 Load Balancing

To evaluate the effect of Clover's load balancing mechanism, we use a synthetic workload with six data entries, a, b, and $c1$ to $c4$. The workload initially creates a (no replication) and b (with 3 replicas) and reads these two entries continuously. At a later time, it creates $c1$ to $c4$ (no replication) and keeps updating them. One CN runs this synthetic workload on three DNs. Figure 13 shows the total traffic to the three DNs with different allocation and load-balancing policies. With a naive policy of assigning DNs to new write requests in a round-robin fashion and reading from the first replica, write traffic spreads evenly across all DNs but reads all go to *DN-1*. With write load balancing, MS allocates free entries for new writes from the least accessed DN. Doing so spreads write traffic more towards the lighter-loaded *DN-2* and *DN-3*. With read load balancing, Clover spreads read traffic across different replicas depending on the load of DNs. As a result, the total loads across the three DNs are completely balanced.

7 Conclusion and Discussion

This paper explores a passive disaggregated persistent memory architecture where remote PM data nodes do not need any processing. We present Clover, a low-cost, fast, and scalable pDPM key-value store which separates data and control planes. We compare Clover with two alternative pDPM systems we built, existing distributed PM systems, and disaggregated systems that include processing at data nodes. We performed extensive evaluation of these systems and learned both the benefits and the limitations of them. We end this paper by discussing our overall findings and our suggestions to future systems builders.

Cost Implication. pDPM's CapEx cost saving compared to aDPM is apparent: pDPM reduces the cost of a processor and hardware (server) packaging to host the processor. pDPM's OpEx cost saving mainly comes from avoiding polling at storage nodes. aDPM needs to busy poll network requests to achieve the low latency that can match PM's performance. Meanwhile, to sustain high-bandwidth network (*e.g.*, 100 Gbps and above), aDPM requires many CPU cores or a parallel-hardware unit like FPGA to poll and process requests in parallel, adding to more runtime cost.

Performance Implication. The major tradeoff of removing computation from storage nodes in pDPM is the potential increase in network RTTs to access storage nodes remotely. While it is generally true that moving computation towards data could achieve good performance, our pDPM key-value store systems demonstrate that with careful design, RTTs in pDPM systems could be minimized and in many cases be the same as aDPM systems. In other cases (*e.g.*, key-value put with high-contention), aDPM has unavoidable performance loss because of extra RTTs. On the other hand, not having enough processing power in aDPM (*e.g.*, when only using ARM-based SoC) could lead to significant performance loss.

Application Implication. Building applications with the pDPM model requires careful design. As we demonstrated with the three different pDPM key-value store systems, different application design choices could directly affect how well pDPM performs and scales. The best design would minimize RTTs while avoiding scalability bottlenecks, as with Clover. This paper focuses on key-value store systems, as they are widely used in many datacenter applications. Other systems such as remote file systems, databases, object stores, and data sharing can also build on the pDPM model. As systems complexity increases, it will be more difficult to optimize pDPM's RTTs with just RDMA read, write, and atomic operation interfaces. We believe that extended RDMA interfaces such as HyperLoop [35] could help in these cases.

Recommendation. This paper explores the extreme of completely removing computation power at storage nodes, which helps set baselines in designing disaggregated PM systems. Going forward, we believe that future disaggregated PM systems would benefit from a hybrid approach. Computation that fundamentally involves multiple data accesses can be moved to storage nodes, while the rest should be kept at compute nodes. Among the former, those that have require high performance can be placed on FPGA or ASIC to avoid high CPU cost, while those that can tolerate degraded performance can be placed at low-power cores.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Anirudh Badam for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 1719215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Private conversation with mellanox sales department, March 2019.

[2] Alibaba Cloud. Super computing cluster. https://www.alibabacloud.com/product/scc, 2018.

[3] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/?nc1=h_ls, 2019.

[4] Amazon. Amazon s3. https://aws.amazon.com/s3/, 2019.

[5] Anton Shilov. Pricing of intel's optane dc persistent memory modules leaks: From $6.57 per gb. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks, 2019. visited on 01/15/20.

[6] Krste Asanović. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers, February 2014. Keynote talk at the 12th USENIX Conference on File and Storage Technologies (FAST '14).

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, UK, June 2012.

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[9] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10gbps line-rate key-value stores with fpgas. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*, San Jose, CA, USA, June 2013.

[10] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS '16)*, London, UK, April 2016.

[11] Brian Cho and Ergin Seyfe. Taking advantage of a disaggregated storage and compute architecture. In *Spark+AI Summit 2019 (SAIS '19)*, San Francisco, CA, USA, April 2019.

[12] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-Defined Network-Attached Key-Value Drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, Providence, RI, April 2019.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, New York, New York, June 2010.

[14] Alexandras Daglis, Dmitrii Ustiugov, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, Taipei, Taiwan, October 2016.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[17] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, July 2019.

[18] Facebook. Introducing bryce canyon: Our next-generation storage platform. https://code.fb.com/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

[19] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, Kartause Ittingen, Switzerland, May 2015.

[20] Gen-Z Consortium, 2018. https://genzconsortium.org.

[21] Google. Available first on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '16)*, Florianopolis, Brazil, August 2016.

[23] Hewlett Packard. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/, 2005.

[24] Hewlett Packard Labs. Memory-driven computing. https://www.labs.hpe.com/memory-driven-computing, 2019.

[25] Huawei. Huawei Launches New-Gen Servers Running

on 2nd-Generation Intel® Xeon® Scalable Processors. https://www.huawei.com/en/press-events/news/2019/4/huawei-new-gen-servers-xeon-scalable-processors.

[26] Intel Corporation. Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html.

[27] Intel Corporation. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, 2019.

[28] Intel Corporation - Product and Performance Information. Intel Non-Volatile Memory 3D XPoint. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html?wapkw=3d+xpoint, 2018.

[29] Intel Corporation - Product and Performance Information. Reimagining the data center memory and storage hierarchy. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/, 2019.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '14)*, Chicago, IL, USA, August 2014.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savanah, GA, USA, 2016.

[33] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, Georgia, USA, April 2016.

[34] Patrick Kennedy. Dual intel xeon e5-2620 (v1, v2 and v3) compared. https://www.servethehome.com/dual-intel-xeon-e5-2620-v1-v2-v3-compared/, 2015.

[35] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, Budapest, Hungary, August 2018.

[36] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, USA, 2019 2019.

[37] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. Sparkle: Optimizing spark for large memory machines and analytics. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[38] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[39] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[40] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Austin, Texas, 2009.

[41] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*, New Orleans, LA, USA, February 2012.

[42] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, CA, USA, July 2017.

[43] Mellanox. ConnectX-6 Single/Dual-Port Adapter supporting 200Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card.

[44] Mellanox. Bluefield smartnic. http://

www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2018.

[45] Mellanox. Mellanox connectx-4 adapter product brief. https://www.mellanox.com/files/doc-2020/pb-connectx-4-vpi-card.pdf, 2020. visited on 06/01/20.

[46] Microsoft. Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/.

[47] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC '13)*, San Jose, CA, USA, June 2013.

[48] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[49] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*, Budapest, Hungary, August 2018.

[50] MongoDB Inc. MongoDB. http://www.mongodb.org/.

[51] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[52] Peng Peng, You Mingyu, and Xu Weisheng. Running 8-bit dynamic fixed-point convolutional neural network on low-cost arm platforms. In *2017 Chinese Automation Congress (CAC)*, Jinan, China, Oct 2017.

[53] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communication of the ACM*, 33(6):668–676, June 1990.

[54] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[56] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[57] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, April 2020.

[58] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 40:40–52, March 2017. Special Issue on Distributed Data Management with RDMA.

[59] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015.

[60] Tom Talpey and Jim Pinkerton. Rdma durable write commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00, 2016.

[61] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA '10)*, Bangalore, India, Jan 2010.

[62] TECHPP. Alibaba singles' day 2019 had a record peak order rate of 544,000 per second. https://techpp.com/2019/11/19/alibaba-singles-day-2019-record/, 2019.

[63] Tejas Karmarkar. Availability of linux rdma on microsoft azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available, 2015.

[64] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. Memory-oriented distributed computing at rack scale. In *Proceedings of the ACM Symposium on Cloud Computing, (SoCC '18)*, Carlsbad, CA, USA, October 2018.

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[66] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[67] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP*

'15), Monterey, CA, USA, October 2015.

[68] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017.

[69] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, February 2019.

[70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[71] YCSB-C, 2015. https://github.com/basicthinker/YCSB-C.

[72] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, 2017.

[73] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores

Alex Conway[*] Abhishek Gupta[†] Vijay Chidambaran[‡] Martin Farach-Colton[§]
Rick Spillane[¶] Amy Tai[||] Rob Johnson[||]

June 5, 2020

[1]

Abstract

Modern NVMe solid state drives offer significantly higher bandwidth and lower latency than prior storage devices. Current key-value stores struggle to fully utilize the bandwidth of such devices. This paper presents SplinterDB, a new key-value store explicitly designed for NVMe solid-state-drives.

SplinterDB is designed around a novel data structure (the STB$^{\varepsilon}$-tree) that exposes I/O and CPU concurrency and reduces write amplification without sacrificing query performance. STB$^{\varepsilon}$-tree combines ideas from log-structured merge trees and B$^{\varepsilon}$-trees to reduce write amplification and CPU costs of compaction. The SplinterDB memtable and cache are designed to be highly concurrent and to reduce cache misses.

We evaluate SplinterDB on a number of micro- and macro-benchmarks, and show that SplinterDB outperforms RocksDB, a state-of-the-art key-value store, by a factor of 6–10× on insertions and 2–2.6× on point queries, while matching RocksDB on small range queries. Furthermore, SplinterDB reduces write amplification by 2× compared to RocksDB.

1 Introduction

Key-value stores form an integral part of system infrastructure. Google's LevelDB [22] and Facebook's RocksDB [8] are widely used, both within their companies and outside. Their importance has spurred research into several aspects of key-value store design, such as increasing write throughput, reducing write amplification, and increasing concurrency [1–3,6,8–17,19,21,22,26,30–32,34,35,37,39,42,43,45–47,50–52].

Existing key-value stores face new challenges with the increasing use of high-performance NVMe solid state drives (SSDs). NVMe SSDs offer high throughput (500K-600K IOPS) and low latency (10-20 microseconds).

LevelDB and RocksDB struggle to utilize all the available bandwidth in modern SSDs. For example, we find that for the challenging but common case of small key-value pairs, RocksDB is able to use only 30% of the bandwidth supplied by an Optane-based Intel 905p NVMe SSD (even when using 20 or more cores).

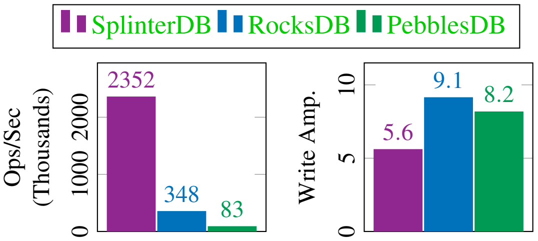


Figure 1: YCSB load throughput and write amplification benchmark results with 24-byte keys and 100-byte values.

We find that the bottleneck has shifted from the storage device to the CPU: reading data multiple times during compaction, cache misses, and thread contention cause RocksDB to be CPU-bound when running atop NVMe SSDs. Thus, there is a need to redesign key-value stores to avoid these CPU inefficiencies.

We present SplinterDB, a key-value store designed for high performance on NVMe SSDs. On workloads with small key-value pairs, SplinterDB is able to fully utilize the device bandwidth and achieves almost 2× lower write amplification than RocksDB (see Figure 1). We show that compared to state-of-the-art key-value stores such as RocksDB and PebblesDB, SplinterDB is able to ingest new data 6–28× faster (see Figure 1) while using the same or less memory. For queries, SplinterDB is 1.5-3× faster than RocksDB and PebblesDB.

Three novel ideas contribute to the high performance of SplinterDB: the ***STB$^{\varepsilon}$-tree***, a new compaction policy that exposes more concurrency, and a concurrent memtable and user-level cache that removes scalability bottlenecks. All three components are designed to enable the CPU to drive high IOPS without wasting cycles.

At the heart of SplinterDB is the STB$^{\varepsilon}$-tree, a novel data structure that combines ideas from log-structured merge trees and B$^{\varepsilon}$-trees. The STB$^{\varepsilon}$-tree adapts the idea of size-tiering (also known as fragmentation) from key-value stores such as Cassandra and PebblesDB and applies them to B$^{\varepsilon}$-trees to reduce write amplification by reducing the number of times a data item is re-written during compaction. The STB$^{\varepsilon}$-tree also introduces a new ***flush-then-compact*** policy that increases

[1] *Rutgers University and VMware Research Group; *Dropbox, Inc.; ‡The University of Texas at Austin and VMware Research Group; §Rutgers University; ¶VMware, Inc.; ||VMware Research Group; {aconway, vchidambaram, robj, rspillane, taiam}@vmware.com; abhi.gupta0290@gmail.com; martin@farach-colton.com

compaction concurrency across the entire tree and exploits locality in the insertion workload to accelerate insertions. By enabling fine-grained, localized compactions, STB$^\varepsilon$-trees push ideas from PebblesDB to their logical conclusion.

Concurrency at the data structural level could be wasted if the data structure is accessed through a cache with poor concurrency. We designed a new user-level concurrent cache for SplinterDB that uses fine-grained, distributed reader-writer locks to avoid contention and ping-ponging of cache lines, as well as a direct map to enable lock-free cache operations. All the data read and written by SplinterDB flows through this concurrent cache.

SplinterDB is not without limitations. Like all key-value stores based on size-tiering, SplinterDB sacrifices the performance of small range queries, although less than one might expect. For large range queries, SplinterDB can use the full device bandwidth. Similarly, size-tiering is known to temporarily increase space usage until multiple versions of a single data item are compacted together. Finally, SplinterDB was designed for the most stringent requirements: small key-value pairs and restricted memory. In cases where key-value pairs are large or memory is plentiful, other choices may prove as good as SplinterDB, and we make some of those comparisons below.

In summary, the contributions of SplinterDB are as follows:

- We introduce the STB$^\varepsilon$-tree, which reduces write amplification and enables fine-grained concurrency in compaction operations (sections 2 to 3).
- We design and build a highly-concurrent memtable that is able to drive enough operations to the underlying STB$^\varepsilon$-tree (section 5).
- We combine the STB$^\varepsilon$-tree, memtable, and user-level cache in SplinterDB, a key-value store that can fully utilize NVMe SSD bandwidth. (section 6).

2 High-Level Design of STB$^\varepsilon$-trees

The basic STB$^\varepsilon$-tree design has three high-level goals:

- Handle inserts using bulk I/O, so that inserts are bandwidth-bound.
- Minimize the number of times each key-value pair gets read or written, so as to reduce write amplifiction, I/O amplification (i.e. read and write amplification), and CPU costs of inserts.
- Maintain sufficient indexing information so that, under normal conditions, each query requires at most one I/O.

In section 3 and section 4, we explain how we refine tree operations to support high concurrency.

The STB$^\varepsilon$-tree shares many ideas with LSM trees, B$^\varepsilon$-trees, and external-memory hash tables from the theory literature. See section 7 for details.

2.1 Overall Structure

The STB$^\varepsilon$-tree is a tree-of-trees, as shown in fig. 2. The backbone of the STB$^\varepsilon$-tree is the ***trunk tree*** (or just trunk). Each node of the trunk has pointers to a collection of B-trees, called ***branch trees*** (or just branches). The branches store all of the actual key-value pairs in the dataset. Each branch also has an associated quotient filter, which serves the same purpose as Bloom filters in LSM trees. Trunk nodes have a fanout of up to F (typically 8 to 16), which is also an upper bound on the number of branch trees. Branch trees have a fanout determined by the number of pivot keys that can be packed into a 4KB node.

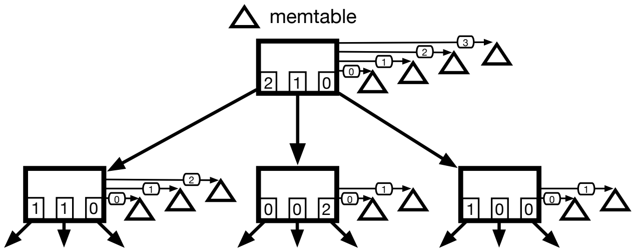


Figure 2: The structure of a STB$^\varepsilon$-tree. The rectangles represent trunk nodes and the triangles represent branch trees and the memtable. The inset boxes indicates the first active branch for each pivot, referencing the pointer labels to the branches.

The overall STB$^\varepsilon$-tree also has a memtable, which is used to buffer insertions, as explained below. The memtable is also a B-tree, just like the branches.

Within a trunk node, the branches are numbered from oldest to youngest, i.e. all the key-value pairs in branch i were inserted before any of the key-value pairs in branch $i+1$. For example, in fig. 2, the root of the trunk tree has four branches, numbered 0 through 3, with branch 0 being the oldest.

Furthermore, stored with each child pointer c in a trunk node is an integer a_c indicating the oldest branch that is ***active*** for that child. Inactive branches are ignored during queries, as explained below. So, for example, in fig. 2, only branches 2 and 3 are active for the root trunk node's leftmost child, branches 1, 2, and 3 are active for its middle child, and all branches (0 through 3) are active for its rightmost child.

2.2 Queries

Queries begin by searching in the memtable. If the queried item is not found in them memtable, then the query proceeds down the trunk tree. Recall that all the data is stored in the branches, and trunk nodes only contain metadata and pointers to branches and filters.

When a query for a key k arrives at a trunk node t, it first searches the pivots of t to determine the correct child c for k. It then iterates over the active branches for c, from youngest to oldest. For each branch b, it first queries b's associated quotient filter. If the quotient filter indicates that k is definitely not in b, then the query moves on to the next branch. Otherwise, is queries for k in b. If it finds a hit, it returns the result to the caller. Otherwise, it moves on to the next branch. If none of the branches contain k, then the query recurses to c.

Analysis. We now explain why queries take at most one I/O in common configurations: those which use at least 32-byte key-value pairs and have RAM which is at least 10% of the dataset size.

The memory used by filters and branch tree indices is bounded as follows. Quotient filters use about 1–2 bytes per key, so the overhead of quotient filters is greatest when key-value pairs are small. With 32-byte key-value pairs, quotient filters will be about 6% of the total database size. Branch trees will have a very high fanout, e.g. ≈ 128 for 4KB nodes, so the interior of the branch trees will be less than 1% of the total database size. Trunk nodes contain only metadata about the branches and filter, and so use negligiable RAM. Therefore all the indexing information will fit comfortably in a RAM that is $\approx 7\%$ of the total database size. For larger keys, e.g. 256 bytes, the branch trees will have a fanout of only about 16, and hence the interior nodes could be up to 6% of the total database size. However, the quotient filters will be less than 1% of the data size, so the indexing data will still be less than 10% of the database size.

Thus the only I/Os during a query will be to load leaves of branch trees. However, the false positive rate of quotient filters with two bytes per key is $< 1\%$, which is low enough to ensure that most queries do not encounter any false positives from the quotient filters that they query. Hence most queries will query exactly one branch tree, which will contain the desired key-value pair. Queries for keys that are not present in the database will usually require no I/Os at all.

2.3 Insertions

When an item is inserted, it is first buffered in the memtable. When the memtable is full, it is added to the root as a new branch, say branch i. We also construct a quotient filter for the memtable at this time. We call the process of adding the memtable to the root of the trunk an ***incorporation***.

The size of the memtable affects performance in the following ways. If the memtable gets too large, then some of its nodes will get evicted from RAM and, once enough nodes spill, a workload of random inserts would require essentially one random I/O per insert, contradicting our goal of handling inserts using bulk I/O. For most systems, this means the memtable should be kept substantially smaller than RAM, e.g. at most a few gigabytes for typical hardware configurations.

On the other hand, the memtable will eventually become a branch, and we want branches to be large enough that scanning a branch can use bulk I/O (i.e. if a branch consisted of only a handful of nodes, then reading the entire branch would be bottlenecked on I/O overheads, rather than bandwidth). Branch scanning performance is critical for compactions and range queries (see Sections 2.4 and 2.6). For most storage devices, it is sufficient to ensure the branches (and hence the memtable) are at least a few megabytes in size.

Thus, for most systems, the memtable can be anywhere from a few megabytes to a few gigabytes in size. Since we are specifically interested in building a system that is robust to low-memory situations, and since making the memtable larger has diminishing returns in terms of scanning throughput, we select a maximum memtable size m that is just comfortably large enough to ensure efficient scanning performance. In our prototype, $m = 24$MB.

2.4 Flushing and Compaction

We cannot keep adding new branches to the root trunk indefinitely. Eventually, the root will fill up and have no more room for branch pointers. This solved by compacting data and flushing it to its children.

This section describes a basic version of flushing and compaction which captures the basic underlying mechanics of a STB$^{\varepsilon}$-tree. In section 3, we describe SplinterDB's more-involved flush-then-compact policy which leverages its B$^{\varepsilon}$-tree structure to expose more compaction concurrency and optimize non-random insertion workloads.

In this simplified version, when a trunk node is full, data is removed from it by repeatedly ***compacting*** some of its branches into a single branch and ***flushing*** the resulting branch to a child until the parent is no longer full (see fig. 3). As in LSM trees, compaction is necessary to keep queries fast. Without compaction, the number of branches that must be queried would grow without bound.

A node p is considered to be full when its branches contain $F \times m$ bytes of active key-value pairs, where F is the fanout and m is the memtable capacity in bytes. When p becomes full, the child c with the most active key-value pairs is chosen and p is flushed into c. We construct a new branch b by compacting all the branches in p that are active for c. Note that the branches in p may contain keys for any of p's children, not just c, so when we compact the branches for a flush to c, we scan over only the portions of each branch that contain keys destined for c. We then add b to c as c's youngest branch. We also build a quotient filter for b and store a pointer to the filter in c. Finally, we mark all the branches in p as inactive for c, so they will not be flushed to c again. Any branches of p that were active only for c are no longer active for any of p's children and can be garbage collected.

Since branches are large, compacting the branches for c can proceed at disk bandwidth. Furthermore, we always flush to the child with the most pending items in the parent. This ensures that the resulting branch b will have at least m items in it, and hence will be efficient to scan when we, at some point in the future, flush it from the child to one of its children.

Analysis. Each time a key-value pair participates in a compaction, it moves one level down the trunk tree. Thus the worst-case write amplification of the STB$^{\varepsilon}$-tree is $O(\log_F N)$, which is the same as in a size-tiered LSM tree. Level-tiered LSM trees and (normal) B$^{\varepsilon}$-trees have a substantially larger write amplification of $O(F \log_F N)$. In Section 3, we describe our flush-then-compact strategy, which enables some key-value pairs to skip compaction at some levels, particularly when the workload exhibits locality.

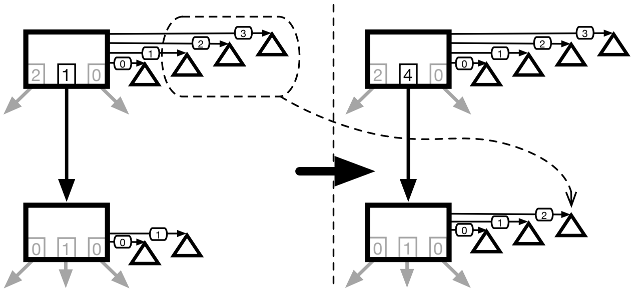


Figure 3: In a STB$^{\varepsilon}$-tree, a flush to a pivot P consists of compacting its active key-value pairs (from its branches) into a new branch in the child. The dashed arrow indicates compaction.

2.5 Splitting

When a trunk leaf is full, it is split, and similarly when a trunk internal node has more than F pivots, it is split. As in standard B-trees and B$^{\varepsilon}$-trees, the I/O costs of splitting and merging do not asymptotically change the costs of inserts. See Section 4 for how we make splitting and merging compatible with hand-over-hand locking in STB$^{\varepsilon}$-trees.

2.6 Iterators and Scans

To construct an iterator starting from key k, we walk the trunk search path for k, constructing B-tree iterators (also starting at k) for each active branch along the path. We then construct a merge iterator on top of the B-tree iterators, which simply returns the smallest key from among all of the underlying B-tree iterators. The merge iterator can be efficiently implemented using a heap.

While constructing the B-tree iterators, we also compute an upper bound u for the leaf of k's search path. As soon as the merge iterator returns a key that is greater or equal to u, we tear down the merge iterator and all the B-tree iterators and rebuild them, starting from u.

2.7 Deletions and Updates

Deletions are implemented through tombstone ***messages***, i.e. a key-value pair with special value indicating that the key has been deleted. More generally, the STB$^{\varepsilon}$-tree supports update messages that encode a function to be applied to the value associated with a key.

3 Flush-then-Compact

This section describes the flush-then-compact algorithm, which improves the I/O and CPU concurrency of compactions during flushing and improves the performance of update workloads with locality, such as sequential workloads. The idea behind flush-than-compact is to decouple the flushing step from the compaction step, as shown in fig. 4.

In a flush, the references to the child's active branches are copied from the parent to the child, along with references to their quotient filters, as shown in fig. 4. The child's active branch counter in the parent is updated to reflect the flush, just as before. At this point the parent and child are in a consistent state and any locks can be released. Since a flush is just a pointer swing, write locks are held very briefly.

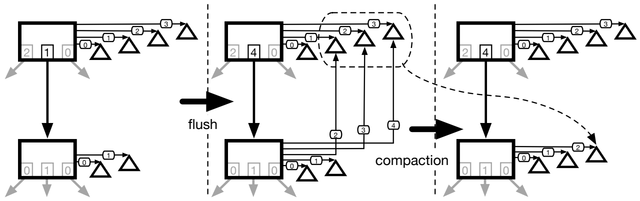


Figure 4: With the flush-then-compact policy, flushes are broken into two steps. First references to the active branches are flushed to the child and removed from the parent (by setting the pivot's active branch number). Then those branches are compacted by an asychronous process. The compaction stage is performed without holding a lock on the node, and during this time it can still flush, be flushed into, split and by queried.

Note that branches can now be referenced by multiple trunk nodes, so branches are reference counted.

From here, if the child is full, we will perform a flush from the child to its children, before initiating any compactions (hence the name "flush-then-compact"). This flush will copy the newly arrived branches from the child to one or more of its children, exactly the same way that the branches were flushed to the child. This process can repeat recursively.

Once all the flushes have completed, we schedule background jobs to compact all the new branches at each node that received a flush. The background jobs will construct the new branches and need to acquire write locks only to replace the old branch pointers with a pointer to the newly created branch.

Flush-then-compact accelerates non-random workloads. Since we perform flushes before compactions, some of the branches involved in a compaction at node p may already be inactive for some of p's children when we perform the compaction. This means we can skip over those keys when we compact those branches. And, since branches are stored as B-trees, we can skip over those key ranges efficiently. Thus we effectively avoid compacting those keys at p's level in the STB$^{\varepsilon}$-tree.

To see why this accelerates non-random insertion workloads, consider an extreme case: a workload of insertions all for a single trunk leaf, ℓ. For simplicity, assume also for the moment that all the nodes on the path from the root to ℓ have zero branches. The memtable will repeatedly fill with key-value pairs for ℓ and get incorporated into the root. Once the root fills, it will flush to its child c along the path to ℓ. This will cause c to immediately become full and flush to its child. This process will repeat until all the branches arrive at ℓ.

At that point, the system will schedule background compactions for each trunk node along the path from the root to ℓ. However, at each node other than ℓ, there will be no live data in any of the branches. The compactions will thus skip all the data in the branches, resulting in empty branches at each interior node. Consequently the interior nodes will again be left with zero branches. Thus only ℓ will have a non-degenerate compaction. As a result, the new key-value pairs will participate in only one compaction, and hence the STB$^\varepsilon$-tree will have a write amplification of 1 for this workload.

Now consider the case when the nodes along the path to ℓ do not have zero branches. The first few times the root fills, it may choose to flush to some child c' that is not along the path to ℓ. This would happen if the root happened to contain more items for c' than c. However, as long as the workload consists only of items for ℓ, eventually the root will contain more items for c than for any of its other children. From that point forward, it will always flush to c. Then the same process will repeat at c. Eventually, each time the root fills, the system will flush all the new branches to ℓ, as described above.

Thus this flushing protocol automatically adapts to the insertion workload without knowing a priori what that workload is. Furthermore, if the workload changes then, after some time, the flushing decisions will adapt to the new workload automatically.

Furthermore, this flushing algorithm exploits less-than-perfect locality automatically. For example, suppose the insertion workload consists almost entirely of key-value pairs for ℓ, but a few random items for other leaves. Almost every time the root fills, it will flush to c, and the process described above will occur. However, each flush will leave a few new items in the root. This cruft will accumulate, eventually causing the root to flush to a child other than c, cleaning out some cruft. After that, the root will resume flushing to c until enough cruft accumulates again. Thus most data will get flushed directly to ℓ, and hence have a write amplification of 1, but a small amount of data will get compacted at every level.

As these examples show, the flush-then-compact algorithm is much more robust than the special-case optimizations frequently implemented for sequential insertions. Special-case optimizations can be foiled by a few random insertions sprinkled into a sequential workload. Flush-then-compact, on the other hand, exploits *locality* rather than *sequentiality*. In section 6.3 we show empirically that flush-then-compact enables SplinterDB to outperform other systems on near-sequential workloads.

Flush-then-compact exposes concurrency. Flush-then-compact improves concurrency by setting up several compactions and then launching them simultaneously, which improves both CPU and I/O parallelism. The hierarchical nature of the B$^\varepsilon$-tree structure makes it trivially safe to perform compactions concurrently at different trunk nodes. In a standard compact-then-flush approach, each time the root is flushed, it initiates a single compaction. The system would not start another compaction until the root is filled (and flushed) again.

4 Preemptive Splitting for STB$^\varepsilon$-trees

Splits and merges pose problems for hand-over-hand locking in B-trees (and B$^\varepsilon$-trees). Hand-over-hand locking proceeds from root to leaf, but splits and merges proceed from the leaves up.

An approach to solving this issue in B-trees is to use preemptive splitting and merging [40]. During a B-tree insert, if a child already has the maximum number of children, then it is split while the insertion thread still holds a lock on its parent. Then the insertion can release the parent's lock and proceed down the tree, assured that the child will not need to split again as part of this insertion. Analogously, deletions merge a child with one of its neighbors if the child has the minimum number of children. This works because insertion and deletions can increase or decrease the number of children of a node by at most 1.

This approach does not work in B$^\varepsilon$-trees, because a flush to a leaf could cause that leaf to split multiple times. In STB$^\varepsilon$-trees with flush-then-compact, we can move all pending messages along a root-to-leaf path to the leaf before performing any compactions, splits, or merges. The total number of messages moved to the leaf is bounded by $O(Fm\log_F N)$, i.e. the capacity of a trunk node times the height of the tree. The leaf can therefore split into as many as $O(\log_F N)$ new leaves of size B. Similarly, a collection of flushes full of delete messages to several leaves of a single parent can reduce the parent's number of children by $O(\log_F N)$. In practice, $\log_F N$ is less than 10 for typical fanouts $F \approx 8$ and dataset size $N \leq 2^{80}$ key-value pairs.

We extend preemptive splitting and merging to STB$^\varepsilon$-trees as follows. We reserve space in each node to accommodate up to $F+H$ children, where H is an upper bound on the tree height, e.g. $H = 10$. We then apply preemptive splitting, except we preemptively split a node during a flush if its fanout is above F. For merges, we take a similar approach. If, during a flush, we encounter a node with less than $F/2$ children, then we merge or rebalance it with one of its siblings.

Thus all operations on the STB$^\varepsilon$-tree—flushes, compactions, splits, and merges—proceed from root to leaf and can therefore use hand-over-hand locking.

The mechanisms for flush-then-compact make it easy to handle branches during splits. Recall that each branch can be marked dead or alive for each child, and branches are refcounted and hence can be shared by multiple trunk nodes. Thus we can split a trunk node by simply giving its new sibling references to all the same branches as the node had before the split. In the new node, we copy the liveness information for each branch along with the children that are moved to the new sibling.

5 From STB$^\varepsilon$-trees to SplinterDB

In this section, we discuss the details of SplinterDB's implementation, which addresses the concurrency and memory bottlenecks associated with driving NVMe devices to full bandwidth.

5.1 Branch Trees and Memtables

SplinterDB uses the same B-tree implementation for both its branches and its memtables, although there are some differences to optimize for their use cases.

Branch trees, extents, and pre-fetching. When a branch is created from a compaction, its key-value pairs are packed into the leaves of the B-tree, and the leading edge of internal nodes are created to index them. The nodes in each level are allocated in extents of 32 pages, and the header of each node stores the address of the following node, but also of the next extent. In this way, the nodes of each level form a singly linked list.

Iteration through a branch is performed by walking the linked list formed by its leaves. Whenever the iterator reaches the beginning of a new extent, it issues an asynchronus prefetch request for the next extent. The extent length is configurable to tune to the latency of the storage device.

Memtables. The basic design of the memtables mirrors that of the branch B-trees, but includes some optimizations to increase their insertion performance and concurrency.

As in the case of the static branch trees, the nodes on each level of the memtable form a singly-linked list, and nodes are allocated in extents. However, because nodes are created on demand as nodes split, we do not try to guarantee that successive nodes reside in the same extent. Furthermore, since memtables are almost always in RAM, we do not perform prefetching during memtable traversals.

The memtable uses hand-over-hand locking together with preemptive splitting, as described by Rodeh [40]. To increase concurrency, write locks are only obtained on internal nodes when a split is required.

To ensure locks are held briefly, especially on nodes near the top of the tree, the tree uses a new technique called ***shadow splitting***. To split a node c, a write lock is obtained on c and the parent p. We allocate a physical block number (PBN) n for the new sibling, c' and add it as part of a new pivot in p. However, in the cache, we initially point n to c. At this point, we can release all locks on p. Now, we fill in the contents of c', update the PBN n to point to c' in the cache, and then release all locks on c'. Finally, we upgrade to a write lock on c, truncate its child list (via a metadata operation) and then release all locks on c.

5.2 User-level Cache and Distributed Locks

SplinterDB has a single user-level cache which keeps recently accessed pages in memory. Almost all the memory that SplinterDB uses comes from this cache, so pages from all parts of the data structure—trunk node pages, branch pages, filter pages and memtable pages—are all stored there. Only cache and file-system metadata, as well as small allocations used to enqueue compaction tasks are allocated from system memory.

This design allows nearly all the free memory to be used for whichever operations are being performed, so that parts of the data structure which are not in use can be paged out.

The cache at a high level is a clock cache, but with several features designed to improve concurrency.

Each thread has a thread-local hand of the clock, which covers 64 pages. The thread draws free pages from the hand, and if it has exhausted them, it acquires a new hand from a global variable using a compare-and-swap. It then writes out dirty pages from the hand which is a quarter turn ahead, and evicts any evictable pages in its new hand. Thus threads clean and evict pages from distinct cache lines within the cache metadata, avoiding contention and cache-line ping-ponging.

SplinterDB uses distributed reader-writer locks [24] to avoid cache-line thrashing between readers. Briefly, a distributed reader-writer lock consists of a per-thread reader counter and a shared write bit. Each reader counter is on a separate cache line to avoid cache-line ping-ponging when readers acquire the lock. Writers set the write bit (using compare and swap) and then wait for all the read counters to become zero. Readers acquire the lock by incrementing their read counter and then checking that the writer bit is 0. If it is not, they decrement their reader counter and restart.

Distributed reader-writer locks allow readers to scale essentially perfectly linearly, at the cost that acquiring a write lock is expensive. However, the design of SplinterDB makes writing rare enough that this is a good trade-off.

We make distributed reader-writer locks space efficent by storing each thread's reader counters in an array indexed by cache-entry index. Each reader counter is one byte, so the total space used by locks is $t \times c$ bytes, where t is the number of threads and c is the number of cache entries.

SplinterDB supports three levels of lock: read locks, "claims", and write locks. A claim is a read lock that can be upgraded to a write lock. Only one thread can hold a claim at a time. After obtaining a read lock, a thread may try to obtain a claim by trying to set a shared claim bit with a test-and-set. If this fails, they must drop the read lock and start over. Otherwise, they can upgrade their claim to a write lock by setting a shared write bit and waiting for all the read counters to go to zero.

5.3 Quotient filters

Bloom filters [7] are the standard filter for most LSMs [8, 22, 39]. However, the cost of Bloom filter insertions can dominate the cost of sorting the data in a compaction. Therefore modern key-value stores often use more efficient filters; for example, RocksDB uses blocked Bloom filters [38];

Similarly, SplinterDB uses quotient filters [4, 5, 36] instead of Bloom filters. A full presentation of quotient filters is out of scope for this paper, but we review their salient features for SplinterDB. See Pandey, et al. for a full presentation on quotient filters [36]. The key feature of quotient filters is that, like blocked Bloom filters, each insert or query accesses $O(1)$ cache lines (and hence $O(1)$ page accesses). Quotient filters are roughly as space efficient as Bloom filters—for the range of parameters used in SplinterDB, quotient filters use between 0.8× and 1.2× the space of a blocked Bloom filter. We view the space as essentially a wash. Quotient filter inserts and lookups also require only one hash function computation. In past work, quotient filter insertions and queries were shown to be 2-4× faster than in a Bloom filter.

A quotient filter for set S stores, without error, $h(S) = \{h(x) \mid x \in S\}$, where h is a hash function. Since the quotient filter stores $h(S)$ exactly, all false positives are the result of collisions under h. Thus each insertion or lookup requires only one hash function computation. Furthermore, a quotient filter stores the elements of $h(S)$ in sorted order in a hash table using a variant of linear probing. Thus most inserts and lookups in a quotient filter access only 1 or 2 adjacent cache lines. As a result, insertions and lookups in quotient filters are typically 2-4× faster than in a Bloom filter. Finally, quotient filters are space efficient, using slightly less space than Bloom filters whenever the false positive rate ε is less than $1/64$, which is typical. For example, a quotient filter with $\varepsilon = 0.1\%$ uses about 10% less space than a Bloom filter [36].

SplinterDB further reduces the CPU costs of filter building during compaction by using a bulk build algorithm. During the merging phase of compaction or when inserting into a memtable, SplinterDB builds an unsorted array of all the hashes of all the tuples compacted or inserted. The array is then sorted (by hash value) and the quotient filter is built. Since the quotient filter also stores the hashes in sorted order, this means that the process of inserting all the hashes is a linear scan of the sorted array and of the quotient filter. Hence it has good locality and can benefit from cache prefetching.

5.4 Logging and Recovery

SplinterDB uses per-thread write-ahead logical logging for recovery. By using per-thread logs, we avoid contention on the head of a single, shared log.

The challenge is to resolve the order of operations across logs after a crash. For this, we use a technique similar to "cross-referenced logs" [25]. Our scheme works as follows. Each leaf of the memtable has a generation number. Whenever a thread inserts a new message into the memtable, it records and increments the generation number of the memtable leaf for the inserted key. It then appends the inserted message to its per-thread log, tagged with the leaf's generation number. During recovery, the generation numbers in the logs give a total order on the operations performed on each leaf (and hence on all the keys for that leaf), so that the recovery procedure can replay the operations on each key in the proper order. When a leaf of the memtable splits, the new leaf gets the same generation number as the old leaf.

6 Evaluation

We evaluate the performance of SplinterDB on several microbenchmarks and on the standard YCSB application benchmark [20]. We compare this performance against that of two state-of-the-art key-value stores, RocksDB and PebblesDB. The following questions drive our evaluation:

- How much does SplinterDB improve insertion performance? To what extent is improvement achieved through reduced write amplification and other factors?
- Despite being size-tired, how much does SplinterDB mitigate [range] query performance? Can SplinterDB utilize device bandwidth for large range queries?
- How much faster are sequential (or otherwise local) insertions in SplinterDB? Do they have lower write amplification?
- Do point lookups scale with the number of threads?

6.1 Setup and Workloads

All results are collected on a Dell PowerEdge R720 with a 32-core 2.00 GHz Intel Xeon CPU, 256 GiB RAM and a 960GiB Intel Optane 905p PCI Express 3.0 NVMe device. The block size used was 4096 bytes.

In general, we use workloads derived from YCSB traces with 24B keys. We generally use 100B values, but also include a set of YCSB benchmarks for 1KiB values. We instrumented dry runs of YCSB in order to collect workload traces for the load and A–F YCSB workloads and replay them on each of the databases evaluated. In order to eliminate the overhead of reading from a trace file during the experiment, the trace replayer `mmaps` the trace file before starting the experiment. We use the same traces for each system.

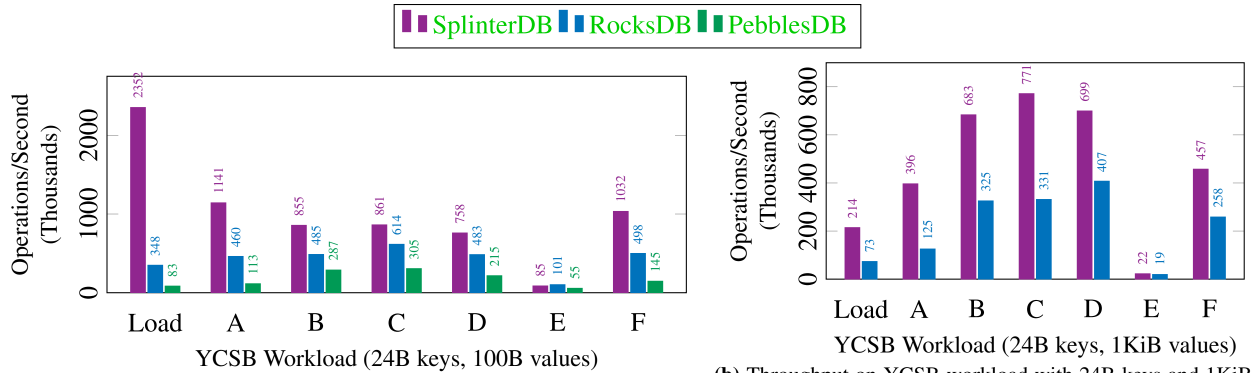
In general, we limit the available memory to 10% of the dataset size or less. In order to perform the benchmarks on reasonably sized datasets, we restrict the available system memory with a type 1 Linux `cgroup`, sized to the target memory size plus the size of the trace, which we pin so that it cannot be swapped out. Unless otherwise noted, the target memory size is 4GiB. PebblesDB has an apparent memory leak, which causes it to consume the available memory, so we allow it to use the full system memory. On the YCSB load benchmarks, this causes it to swap for a small portion at the end, but this was less than 10% of the run time.

Unless otherwise noted, SplinterDB uses a max fanout of 8, a memtable size of 24MiB and a total cache size of 3.25GiB. The difference between this cache size and the target memory size of 4GiB is to accommodate other in-memory data structures maintained by SplinterDB.

Each system is run with the thread count which yields the highest throughput. RocksDB is configured to use background threads equal to the number of cores minus the number of foreground threads, with a minimum of 4. PebblesDB uses its default number of background compaction threads. SplinterDB is configured without background compaction threads.

6.2 YCSB

We measure application performance using the Yahoo Cloud Services Benchmark (YCSB). The core YCSB workloads consist of load phases and run phases. The load phases create a dataset by inserting uniformly random key-value pairs. The run phases emulate various workload mixes. Workload A is 50% updates, 50% reads, workload B is 95% reads, 5% updates), workload C is 100% reads, workload D is read latest (95% reads, 5% insertions), workload E is short range scans (95% scans, 5% insertions) and workload F is read-modify-writes (50% reads, 50% RMWs).



(a) Throughput on YCSB workloads with 24B keys and 100B values. Load is 673M operations, E is 20M operations and others are 160M operations. Higher is better.

(b) Throughput on YCSB workload with 24B keys and 1KiB values. Load is 84M operations, E is 1.3M operations and others are 10M operations. Higher is better.

Figure 5: YCSB throughput and I/O benchmark results.

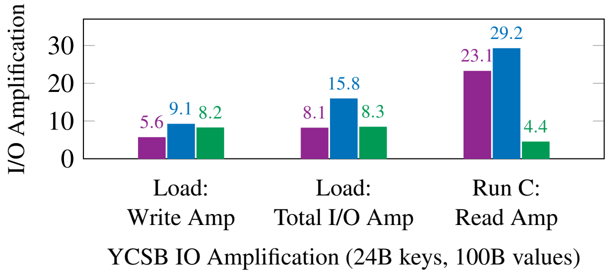


Figure 6: IO amplification on YCSB load and Run C workloads, as measured with iostat. Lower is better.

system	mean	median	P95	P99
SplinterDB	7.0	3.1	12.4	27.7
RocksDB	40.2	29.7	50.5	86.7

Table 1: Insertion latency (μs) for the workload in fig. 5a.

system	mean	median	P95	P99
SplinterDB	46.4	13.3	126.3	216.1
RocksDB	51.1	28.8	108	221.1

Table 2: Read latency (μs) for the workload in fig. 5a.

The results with 100B values and 1KiB values are shown in Figure 5 (both workloads use 24B keys). Figure 6 shows the write and I/O amplification in the 100B-value benchmark.

On the load phase, SplinterDB is faster than RocksDB by almost an order of magnitude. Because of size-tiering and its compaction/flushing policy SplinterDB has about 1/2 the write amplification of the other systems. Note PebblesDB performs almost no reads because it was given unlimited memory. Surprisingly PebblesDB does not show substantially lower write amplification than RocksDB.

On the run phases, which the exception of E, SplinterDB is 40–150% faster than RocksDB, the next fastest system. On E, SplinterDB is roughly 15% slower than RocksDB in the 100B-value case, and about 15% faster than RocksDB in the 1KiB-value benchmark.

Latency. SplinterDB maintains high throughput without sacrificing latency. Table 1 reports insertion latency for SplinterDB and RocksDB. Unsurprisingly, the latency of RocksDB is at least 3x that of SplinterDB on all metrics. This is because mechanisms such as flush-and-compact (Section 3) improve concurrency and eliminate stalls on the write path.

Table 2 reports read latency for SplinterDB and RocksDB. SplinterDB read latency is comparable to RocksDB, because the quotient filters (section 5.3) in SplinterDB behave similarly to Bloom filters in RocksDB.

KVell. KVell [33] is a key-value store also designed to utilize full NVMe bandwidth. It has an in-memory B-tree index that maps all keys to disk page offsets. It does well on large (4KiB) key-value pairs, but on small key-value pairs, the overhead of the in-memory index becomes a significant fraction of the dataset size. In particular, it was impossible to run KVell in a memory `cgroup` of 4GiB. Figure 7 shows KVell's performance on the YCSB workload with 100B values, for different memory sizes. At 22GiB, which is around the size of the in-memory index, KVell's performance starts to drop. At 20GiB, KVell becomes unusable. Therefore in realistic memory settings, KVell is not a viable option for the small key-value sizes that SplinterDB targets.

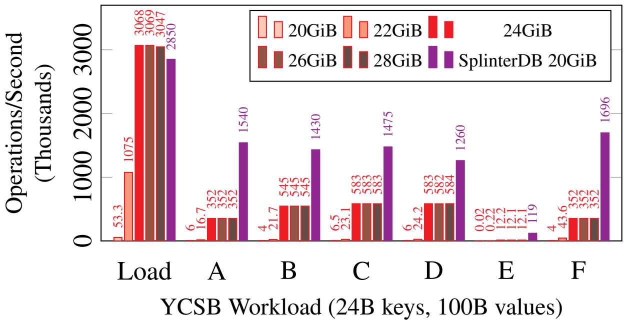


Figure 7: Throughput of Kvell on YCSB workloads with varying amounts of available RAM, 100B values. Throughput of SplinterDB with 20GiB RAM shown for comparison. Load consists of 673M operations, E consists of 20M operations and all other workloads consist of 160M operations. Higher is better.

SplinterDB is designed to work well even under low-memory scenarios (less than 10% of total data size). However, we also run the YCSB experiment with higher memory, 20 GiB, to compare with KVell in a regime where KVell per-

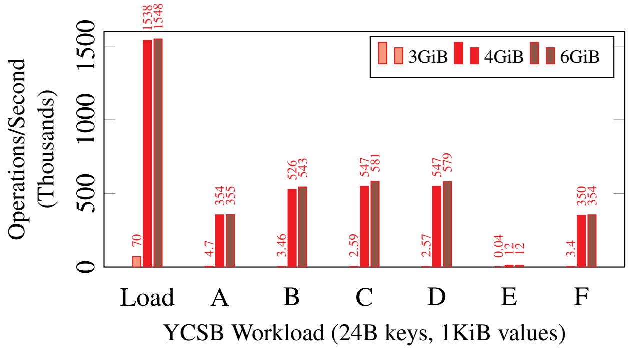


Figure 8: Throughput of Kvell on YCSB workloads with varying amounts of available RAM, 1 KiB values. Load consists of 84M operations, E consists of 1M operations and all other workloads consist of 10M operations. Higher is better.

forms well. As shown in fig. 7, we find that SplinterDB almost matches KVell on insertions, but outperforms KVell by roughly a factor of 2.5 on queries.

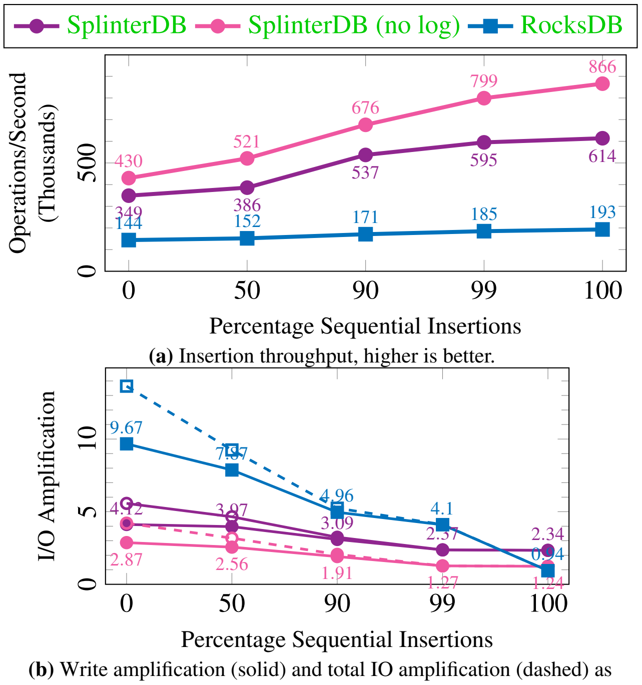
For larger values, the memory cliff for KVell is much lower. We run the same YCSB workload on both systems, but with 1KiB values. In this case, KVell's memory cliff is between 3GiB and 4GiB, as shown in fig. 8. For these larger values, KVell outperforms SplinterDB insertions (see Figure 5b) due to a low write amplification, but still can only achieve 55-77% query throughput of SplinterDB on the other YCSB workloads. KVell's range-query performance (workload E) is particularly lower than SplinterDB's because KVell does not keep key-value pairs sorted. Thus each 1KiB key-value pair in the range requires a separate, random 4KiB I/O, resulting in a read amplification of about 4×. Splinter, on the other hand, sorts and packs key-value pairs into 4KB blocks, for a read amplification close to 1 during range queries.

As soon as the memory cliff hits, KVell exhibits the same performance drop as in the previous experiment. However, when values are so large, this may not be so important, since indexing information can easily fit in RAM.

6.3 Sequential Insertion Performance

Because of the flush-then-compact policy, we expect SplinterDB's performance will improve substantially on insertion workloads with a high degree of locality (see section 3). We demonstrate this by performing 20GiB of single-threaded insertions from a trace composed of interleaved sequential and random keys in different proportions. For comparison, we perform the same workload on RocksDB.

As shown in fig. 9a, SplinterDB's performance improves smoothly from 349K insertions per second for a purely random workload to 614K insertions per second for a purely sequential workload, which is 76% faster. This improvement is partially obscured by the log, which adds a constant additive IO overhead. If we disable the log, SplinterDB improves from 430K insertions per second on a purely random workload to 866K operations per second on a purely sequential workload, 100% faster. Note that we would expect the intermediate



(a) Insertion throughput, higher is better.

(b) Write amplification (solid) and total IO amplification (dashed) as measured with `iostat`. Lower is better.

Figure 9: Single-threaded insertion throughput by varying mixed sequential/random locality percentage. X-axis not to scale.

throughputs in the best case to be the [weighted] harmonic mean of the pure cases, because they are rates. At 50% random, 50% sequential for SplinterDB with no log this is 575K insertions/second, so its actual performance of 521K insertions/second captures a substantial amount of the potential improvement.

RocksDB also improves as the workload becomes more sequential, but this effect is much smaller, a 35% speedup. Furthermore, RocksDB shows less than 20% speedup until the workloads becomes 99% sequential.

Figure 9b shows that as predicted, SplinterDB incurs less IO amplification on more sequential workloads. With the log disabled, its write amp approaches 1 as the workload approaches purely sequential. In contrast, while RocksDB also has less IO amplification on more sequential workloads, it still incurs write amplification of 4.1 even when 99% of the keys are sequential. It is only when the workload becomes 100% sequential that the write amplification becomes close to 1 (because of caching it even falls below 1).

6.4 Concurrency Scaling

SplinterDB is designed to scale with the number of available cores up to the performance limits of the storage device. This is especially true for reads, where the use of distributed reader-writer locks and a highly concurrent cache design, together with a careful avoidance of dirtying cache lines, can avoid almost all contention between threads.

Read Concurrency. We test the read concurrency scaling of SplinterDB by running YCSB workload C with 160M key-value pairs, where, as in fig. 5a each instance of the test divides

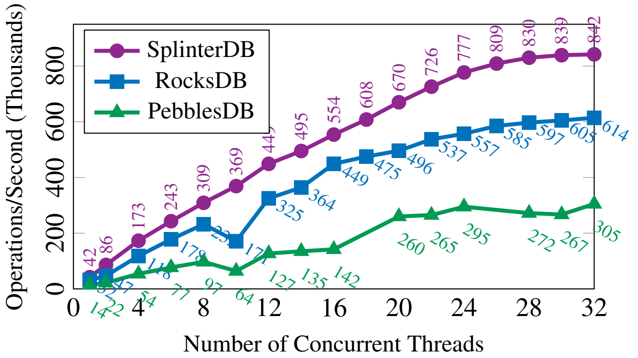


Figure 10: Read concurrency: read throughput (YCSB workload C) by number of threads. Each instance performs 160M reads divided evenly between threads. Higher is better.

the keys evenly into *N* batches, which are then performed in parallel by *N* threads. The results are in fig. 10.

The results show nearly linear scaling—throughput with 24 threads is 18.5× the single-threaded throughput. Between roughly 24 and 32 threads, the scaling flattens out, but at that point the measured throughput is 2.07–2.24 GiB/sec, which is 88–95% of the device's advertised random read capability.

While RocksDB also scales well, its throughput with 24 threads is 17.4× its single-threaded throughput, and with 32 threads it uses 91% of the device's advertized random read capability. Therefore, even though SplinterDB can perform more operations per second, RocksDB is still making nearly full use of the device for reads. We conclude here that SplinterDB is making better use of the available memory for caching, since it has noticeably lower read amplification. Finally, PebblesDB is unable to scale with more threads, flattening out at around 300K reads/sec.

Insertion Concurrency We test the insertion concurrency scaling of SplinterDB by running the YCSB load workload with 673M key-value pairs divided into *N* batches, each of which is inserted in parallel by a different thread. fig. 11 reports throughput for various *N*.

The results show that SplinterDB scales almost linearly up to 10 threads. With 10+ threads, it performs 2.0-2.4M insertions per second with IO amplification around 7.5, which implies that it uses 1.9-2.2GiB/sec of bandwidth, which is at or near the device's sequential bandwidth of 2.2GiB/sec.

RocksDB's insertion performance also scales as the number of threads increase up to 14 threads, by a factor of 2.7. At its peak, it uses 754GiB/sec of bandwidth. PebblesDB scales slightly as well. For both RocksDB and PebblesDB, as many background threads as available are used for flushing and compaction during this benchmark.

6.5 Scan Performance

An inherent disadvantage of size-tiering is that short scans must search every branch along the root-to-leaf path to the starting key. Each of these searches is likely to incur an IO to the device. As a result, as seen in fig. 5a, SplinterDB with 124B key-value pairs has scan throughput on small ranges that is about 85% that of RocksDB. During that workload, SplinterDB performed 2.26 GiB/sec of IO, which is within 96% of the devices advertised random read capability (short scans of small key-value pairs are essentially random reads).

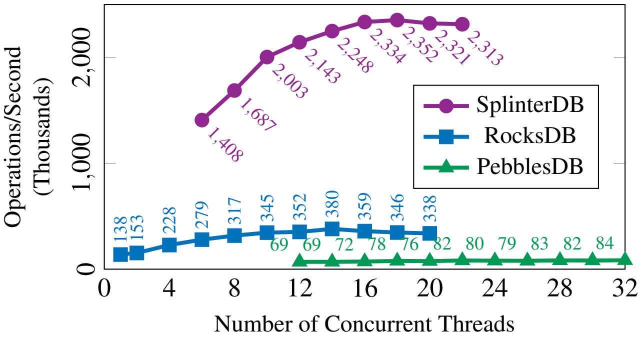


Figure 11: Insertion concurrency: insert throughput (YCSB Load) by number of threads. Each instance performs 673M writes divided evenly between threads. Higher is better.

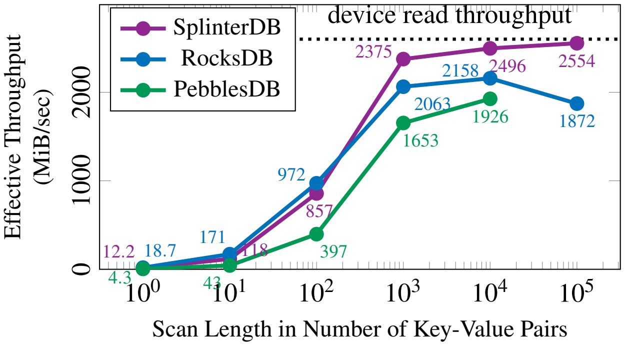


Figure 12: Scan throughput in MiB/sec as a function of scan length. For small scans, the start up cost dominates, but as the scans get longer, the throughput approaches the device's advertised bandwidth (2.6GiB/sec). The x-axis is on a log scale. Higher is better.

However, once the initial search for the successor to the starting key has completed, the root-to-leaf path within each relevant branch will be in memory. Together with prefetching, this allows subsequent keys to be fetched at near disk bandwidth. Therefore, we expect that scans have a relatively high startup cost for the search to the starting key, followed by a very low iteration cost of obtaining subsequent keys.

Thus, when the amount of data requested grows to multiple pages, the disadvantage begins to dissipate. One way this happens is with larger key-value pairs: with 1kib values, SplinterDB is about 16% faster than RocksDB.

Another way this can happen is with scans of more key-value pairs. We modify YCSB workload E to have only fixed-length scans of *N* key-value pairs, where *N* is 1, 10, 100, 1K, 10K or 100K. We perform runs of 10M scans of length 1, 10 and 100, 1M scans of length 1000, 100K scans of length 10000 and 10K scans of length 100000. Each run is performed on a dataset of 80GiB (with 24B keys and 100B values) and 4GiB memory.

The result is shown in fig. 12. Short scans on SplinterDB have low effective bandwidth, and in fact the bandwidth scales close to linearly with the scan length for scans of up to 100 key-value pairs. This suggests that for scans of this length, the startup cost dominates the iteration cost, which is as expected. As the scan length increases, the effective bandwidth of the scans approaches the device's advertised sequential read bandwidth, delivering 91% at scans of 1,000 key-value pairs. At scans as small as 100 key-value pairs, SplinterDB returns data at nearly half the bandwidth of the device.

7 Related Work

The STB$^\varepsilon$-tree is based on a B$^\varepsilon$-tree, a data structure that has been used in several file systems and databases [18, 27–29, 44, 48, 49]. The closest work to ours is Tucana [37], a B$^\varepsilon$-tree optimized for SSDs. They also focus on CPU cost, concurrency, and write amplification. Our work pushes this to the even more demanding case of NVMe devices. SplinterDB improves on techniques that have been applied to log-structured merge (LSM) trees and key-value stores to reduce write amplification and increase concurrency.

Size-Tiering. Cassandra [19], Scylla [42] PebblesDB [39], and RocksDB [8] (in "universal compaction" mode) use size-tiering to reduce write amplification. Size tiering delays compaction of sorted runs in order to reduce write amplification. This can harm query performance because queries must search more runs to find the queried item. Fluid LSMs [16], Dostoevsky [16], LSM bushes [17], and Wacky [17] use hybrids between size-tiering and level-tiering to tune the trade-off between write amplification and query performance. See [39] for a survey of LSM-compaction schemes.

Size-tiering also decreases write amplification in SplinterDB. Because of the design of the STB$^\varepsilon$-tree, SplinterDB further leverages size-tiering for flush-and-compact, which greatly increases the concurrency of background operations.

Write amplification vs. range queries. Several systems sacrifice range-query performance in order to reduce write amplification in other ways. Wisckey [34] reduces write amplification by declustering their key-value store: they log values and only store keys in the LSM-Tree. Since values are stored on disk in arrival order, a range query must gather values from the log. On NVMe, this is not a problem once the values are 4KB or larger. However, for smaller values, this can induce huge read amplification, limiting range query performance to a tiny fraction of device bandwidth. HashKV [10] builds on Wisckey by introducing hash-based data-grouping to further reduce write amplification, but inherits Wisckey's range query performance limitations.

Other approaches improve write amplification by sacrificing range queries altogether. Conway et al. [14] describe a write-optimized hash table, called the BOA, that also uses size-tiering with an LSM. They also introduce the concept of a routing filter, which extends the functionality of Bloom filters, in order to speed up queries. The principle advantage of routing filters is that performance does not degrade as much when they don't fit in RAM. The BOA meets a provable lower bound on the I/O costs of insertions and queries [26]. The downside is that the BOA does not support range queries, which are crucial to many key-value-store applications. LSM-tries [46] organize the LSM tree using tries, resulting in reduced write amplification. However, LSM-tries do not support range queries.

Other approaches. Researchers have also attempted to reduce write amplification by exploiting special hardware features such as flash translation layers [35] and vector interfaces [45]. VT-Tree [43] uses indirection to avoid copying data that is already sorted, similar to "trivial moves" in RocksDB and PebblesDB. TRIAD [1] reduces write amplification by holding hot keys in memory, delaying compaction until different runs have significant key overlap, and by reducing redundancy between log and LSM tree writes. All these techniques are orthogonal to our work and can be used in conjunction with our techniques.

Concurrency is also an important aspect of key-value store performance. One of the first works in increasing concurrency in LSM-based stores was cLSM [21] which introduces a new compaction algorithm. Zuo et al. [52] show how to tune a cuckoo hash for NVM. Such a scheme suffers from high write amplification, since each insertion must re-write all keys in a data block. Zuo et al. do not report write amplification numbers but instead focus on concurrency.

Kourtis, et al. describe several systems-level optimizations for improving key-value-store throughput on NVMe, such as efficient use of user-level asynchronous I/O and low-latency scheduling [31]. Their techniques are largely orthogonal to the work in this paper.

8 Analysis

We begin with a disk-space analysis, showing that, in STB$^\varepsilon$-tree, size-tiered compaction and flush-then-compact do not blow up the on-disk space usage by more than a constant factor. We then use this to analyze memory usage from indexes and filters, and finally summarize STB$^\varepsilon$-tree's asymptotic performance.

Disk-space. Like level-tiered and size-tiered LSM trees and B$^\varepsilon$-trees, the STB$^\varepsilon$-tree can have a space overhead when there are updates to existing keys. This is because all of these data structures buffer updates and apply them lazily. We begin by showing that the space used by the STB$^\varepsilon$-tree is $O(N)$, where N is the number of distinct keys in the database. This compares quite favorably to the space of a size-tiered LSM, which can be as bad as $\Theta(FN)$.

Theorem 1. *Let N be the number of distinct keys in a STB$^\varepsilon$-tree. Then the STB$^\varepsilon$-tree uses $O(N)$ space on disk.*

Proof. We give only a sketch. The four key observations in the proof are that (1) every leaf must be at least half full of distinct keys due to the splitting and compaction policy, (2) each branch has size at most mF due to the flushing policy, (3) each non-leaf trunk node references at most $3F$ branches due to the flushing policy, and (4) the number of non-leaf trunk nodes is at most $O(1/F)$ times the number of leaves. Together, these prove that the total amount of data referenced in the interior of the tree is at most a constant factor times the number of distinct keys in the leaves.

□

For a workload of random updates to existing keys, we estimate that the space blowup would be roughly a factor of 3. If the workload also contains insertions of new keys, then the blowup would be even lower.

Asymptotic analysis. The height of the trunk is $O(\log_F N/Fm)$, and each item gets compacted at most once per level, so the I/O complexity of random insertions are $O(\frac{\log_F N/Fm}{B})$, which is the same as in a size-tiered LSM tree.

Assuming that all index nodes and filters fit in RAM, the I/O complexity of random point queries is $O(1)$ I/Os, since the filters will eliminate all but the correct branch from being searched.

Long sequential insertion workloads will cost $O(1/B)$ I/Os per item. The I/O efficiency comes from the fact that, once the first batch of items gets flushed to a leaf, the root-to-leaf path for future insertions will be in cache, so no more I/O will be needed, except to write out the new data. This also workload has $O(1)$ pass complexity because our flush-then-compact policy will skip compactions at intermediate layers. A straightforward implementation of a size-tiered LSM, on the other hand, will have the same I/O and pass complexity for both random and sequential insertion workloads.

Range queries returning k items cost $O(F \log_F N/Fm)$ I/Os to get started (since the range query must perform a query in every branch along the root-to-leaf path of the query key). Thereafter, they cost $O(k/B)$ I/Os to return all the items. This is comparable to the I/O cost of range queries in a size-tiered LSM tree.

9 Conclusion

Our work shows that, by combining ideas from LSM trees and B$^\varepsilon$-trees, we can build a key-value store that outperforms current key-value stores by up to an order of magnitude on insertions, matches or outperforms on lookups, and is competitive on range queries.

SplinterDB targets the common case of small key-value pairs and non-uniformly random workloads. Many real-world key-value workloads come from different clients, some of which might be performing very localized operations, while others are performing relatively random operations. SplinterDB exploits whatever locality is available.

SplinterDB makes contributions to both the data-structural and systems design of high-performance key-value stores. We show how to get the low write amplification of size-tiered data structure while maintaining the high query throughput and workload-adaptivity of a B$^\varepsilon$-tree. We also describe several systems issues, such as cache, lock, and memtable design, that one must address to extract the full performance of high-performance NVMe devices.

10 Acknowledgements

We would like to thank Ittai Abraham for his insight and contribution to this project.

We would also like to thank the anonymous reviewers and our shepherd, Ashvin Goel, for their insightful comments.

References

[1] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In Dilma Da Silva and Bryan Ford, editors, *USENIX ATC*, pages 363–375. USENIX Association, 2017.

[2] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In Tova Milo and Wang-Chiew Tan, editors, *SIGMOD*, pages 289–302. ACM, 2016.

[3] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA*, pages 81–92. ACM, 2007.

[4] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012.

[5] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. In Irfan Ahmad, editor, *HotStorage*. USENIX Association, 2011.

[6] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *SIGMOD*, pages 69–78. ACM, 2017.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] Dhruba Borthakur. Rocksdb github wiki – performance benchmarks, 2013.

[9] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 546–554. ACM/SIAM, 2003.

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In Gunawi and Reed [23], pages 1007–1019.

[11] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. How to fragment your file system. *login Usenix Mag.*, 42(2), 2017.

[12] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In Daniel Peek and Gala Yadgar, editors, *HotStorage*. USENIX Association, 2019.

[13] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In Geoff Kuenning and Carl A. Waldspurger, editors, *USENIX FAST*, pages 45–58. USENIX Association, 2017.

[14] Alexander Conway, Martin Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In Salihoglu et al. [41], pages 79–94.

[16] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *SIGMOD*, pages 505–520. ACM, 2018.

[17] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *SIGMOD*, pages 449–466. ACM, 2019.

[18] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In Raju Rangaswami, editor, *HotStorage*. USENIX Association, 2012.

[19] Apache Software Foundation. Apache Cassandra, 2019.

[20] Steffen Friedrich and Norbert Ritter. YCSB. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[21] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys 15)*, page 32. ACM, 2015.

[22] Inc. Google. Leveldb, 2019.

[23] Haryadi S. Gunawi and Benjamin Reed, editors. *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018.

[24] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *IPPS*, 1992.

[25] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In Gunawi and Reed [23], pages 967–979.

[26] John Iacono and Mihai Patrascu. Using hashing to solve the dictionary problem (in external memory). *CoRR*, abs/1104.2799, 2011.

[27] William Jannen, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Lazy analytics: Let other queries do the work for you. In Nitin Agrawal and Sam H. Noh, editors, *HotStorage*. USENIX Association, 2016.

[28] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In Jiri Schindler and Erez Zadok, editors, *USENIX FAST*, pages 301–315. USENIX Association, 2015.

[29] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *TOS*, 11(4):18:1–18:29, 2015.

[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, 2019. USENIX Association.

[31] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, 2019. USENIX Association.

[32] Bredley Kuszmaul. Tokutek White Paper: A Comparison Of Log-Structured Merge (LSM) And Fractal Tree Indexing, 2014.

[33] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In Tim Brecht and Carey Williamson, editors, *SOSP*, pages 447–461. ACM, 2019.

[34] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, 2016.

[35] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: a scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, 2015.

[36] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In Salihoglu et al. [41], pages 775–787.

[37] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[38] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[39] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[40] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 2008.

[41] Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors. *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017.

[42] Inc. Scylla. ScyllaDB: The real-time big data database, 2019.

[43] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, 2013.

[44] Tokutek, Inc. TokuDB, 2014. http://www.tokutek.com.

[45] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC 12)*, page 8. ACM, 2012.

[46] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association, 2015.

[47] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on HM-SMR drives with gear compaction. In Arif Merchant and Hakim Weatherspoon, editors, *USENIX FAST*, pages 159–171. USENIX Association, 2019.

[48] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In Angela Demke Brown and Florentina I. Popovici, editors, *USENIX FAST*, pages 1–14. USENIX Association, 2016.

[49] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *TOS*, 13(1):3:1–3:26, 2017.

[50] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In Nitin Agrawal and Raju Rangaswami, editors, *USENIX FAST*, pages 123–138. USENIX Association, 2018.

[51] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In Sam H. Noh and Brent Welch, editors, *USENIX FAST*, pages 75–89. USENIX Association, 2020.

[52] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *OSDI*, pages 461–476. USENIX Association, 2018.

Twizzler: a *Data-Centric* OS for Non-Volatile Memory

Daniel Bittman
UC Santa Cruz

Peter Alvaro
UC Santa Cruz

Pankaj Mehra
IEEE Member

Darrell D. E. Long
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

Abstract

Byte-addressable, non-volatile memory (NVM) presents an opportunity to rethink the entire system stack. We present Twizzler, an operating system redesign for this near-future. Twizzler removes the kernel from the I/O path, provides programs with memory-style access to persistent data using small (64 bit), object-relative cross-object pointers, and enables simple and efficient long-term sharing of data both between applications and between runs of an application. Twizzler provides a clean-slate programming model for persistent data, realizing the vision of Unix in a world of persistent RAM.

We show that Twizzler is simpler, more extensible, and more secure than existing I/O models and implementations by building software for Twizzler and evaluating it on NVM DIMMs. Most persistent pointer operations in Twizzler impose less than 0.5 ns added latency. Twizzler operations are up to 13× faster than Unix, and SQLite queries are up to 4.2× faster than on PMDK. YCSB workloads ran 1.1–2.9× faster on Twizzler than on native and NVM-optimized SQLite backends.

1 Introduction

Byte-addressable non-volatile memory (NVM) on the memory bus with DRAM-like latency [23, 38] will fundamentally shift the way that we program computers. The two-tier memory hierarchy split between high-latency persistent storage and low latency volatile memory may evolve into a single level of large, low latency, and directly-addressable persistent memory. Mere incremental change will leave dramatic improvements in programmability, performance, and simplicity on the table. It is essential that operating systems and system software evolve to make the best use of this new technology.

These opportunities motivate us to revisit how programs operate on persistent data. The separation of volatile memory and high-latency persistent storage at the core of OS design requires the OS to manage ephemeral copies of data and interpose itself on persistence operations, a penalty that will consume an increasing fraction of time as NVM performance increases [64]. The direct-access nature of NVM invites the use of load and store instructions to directly access persistent data, simplifying applications by enabling persistent data manipulation without the need to transform it between in-memory and on-storage data formats. Thus, the model that best exploits the low latency nature of NVM is one in which persistent data is maintained as in-memory data structures and not serialized or explicitly loaded or unloaded. To avoid serialization, this model must support *persistent pointers* that are valid in *any* execution context, not just the one in which they were created.

Trying to mold NVM into existing models will not enable its fullest potential, just as SSDs did not reach their full potential until they transcended the disk paradigm. To explore a "clean-slate" approach, we are building Twizzler, an OS designed to take full advantage of this new technology by rethinking the abstractions OSes provide in the context of NVM. Twizzler divides NVM into *objects* within a global object space, and pointers are interpreted in the context of the object in which they reside. This decouples pointers from the address space of an individual thread, providing a data-centric programming model rather than a process-centric one. The result is a vastly simpler environment in which the OS's primary function is to support manipulating, sharing, and protecting persistent data using few kernel interpositions.

We implemented a simple, standalone kernel that supports a userspace for NVM-based applications, with compatibility layers for legacy programs. We wrote a set of libraries and portability layers that provide a rich environment for applications to access persistent data that takes into account both semantics (persistent pointers) and safety (building crash-consistent data structures). We then performed a case-study by writing software for Twizzler, taking into account the new flexibility and power gained by our model and evaluating our software for complexity and performance. We ported SQLite to Twizzler, showing how our approach can provide significant performance gains on existing applications as well.

In a world where in-memory data can last forever, the context required to manipulate that data is best coupled with the *data* rather than the process. This key insight manifests itself in the three primary contributions of this paper:

- We discuss (§ 2) our vision for a data-centric OS and the requirements that it must meet to provide low latency memory-style access to NVM with efficient data sharing.
- We present Twizzler (§ 3) and describe its mechanisms to meet those requirements, including decoupling traditionally linked concerns, reducing kernel involvement in address space management, and providing a rich model for constructing in-memory persistent data structures that can be easily shared between programs and machines.
- We evaluate (§ 4) the ease-of-use, security advantages, and programmability offered by our environment, for both new and existing, ported software (SQLite), along with performance improvements (§ 5) on NVM DIMMs.

2 The Data-Centric OS

Operating systems provide abstractions for data access that reflect the hardware for which they were designed. Current I/O interfaces and abstractions reflect the structure of mutually exclusive volatile and persistent domains, the hallmarks of which are heavy kernel involvement for persisting data, a need for data serialization, and complexity in data sharing requiring the overhead of pipes or the management cost of shared virtual memory. However, the introduction of low latency and directly attached NVM into the memory hierarchy requires that we rethink key assumptions such as the use of virtual addresses, the kernel's involvement in persistent I/O, and the way that programs operate on and share persistent data [30].

The first key characteristic of NVM is low latency: only 1.5–8× the latency of DRAM in most cases [38], so the cost of a system call to access NVM dominates the latency of the access itself. The second key characteristic is that the processor can directly access persistent storage using load and store instructions. Direct, low latency access to NVM means that explicit serialization is a poor fit—it adds complexity, as programmers must maintain different data formats and the transformations between them, and the overhead is intolerable due to NVM's low latency. Hence, we should design the semantics of the programming model around *in-memory* persistent data structures, giving programs direct access to them without explicit persistence calls or serialization methods.

These characteristics imply two basic requirements for OSes to most effectively use NVM:

1. **Remove the kernel from the persistence path.** This addresses both characteristics. System calls to persist data are costly; we must provide lightweight, direct, memory-style access for programs to operate on persistent data.
2. **Design for pointers that last forever.** Long-lived data structures can directly reference persistent data, so pointers must have the same lifetime as the data they point to. Virtual memory mappings are, by contrast, ephemeral and so cannot effectively name persistent data. Persistent data is, by definition, accessed by multiple actors, both simultaneously and over time, and thus must be stored in a form that is conducive to sharing without needing the ephemeral context associated with a particular actor.

We call an OS that meets both of these requirements *data-centric*, as opposed to current OSes, which are *process-centric*. Operations on persistent, in-memory data structures are the primary functions of a data-centric OS, which tries to avoid interposing on such operations, preferring instead to intervene only when necessary to ensure properties such as security and isolation. To meet both of these requirements a data-centric OS must provide effective abstractions for identifying data independent of data location, constructing persistent data relationships that do not depend on ephemeral context, and facilitating sharing and protection of persistent data.

2.1 Existing Interfaces

Current OS techniques do not meet these requirements—file `read` and `write` interfaces, designed for sequential media and later expanded for block-based media, require significant kernel involvement and serialization, violating both requirements. While support for these interfaces can be useful for legacy applications, as we will demonstrate, providing the programmer with abstractions designed *for* NVM both reduces complexity and improves performance.

The `mmap` call attempts to hide storage behind a memory interface through hidden data copies. But, with NVM, these copies are wasteful, and `mmap` still has significant kernel involvement and the need for explicit `msync` calls. "Direct Access" (DAX) tries to retrofit `mmap` for NVM by removing the redundant copy, but this fails to address requirement two! Operating on persistent data through `mmap` requires the programmer to use either fixed virtual addresses, which presents an infeasible coordination problem as we scale across machines, or virtual addresses directly, which are ephemeral and require the context of the process that created them.

Attempting to shoehorn NVM programming atop POSIX interfaces (including `mmap`) results in complexity that arises from combining multiple partial solutions. Given some feature desired by an application, the NVM framework can provide an integrated solution that meshes well with the existing support for persistent data structure manipulation and access, or it can fall-back to POSIX resulting in the programmer needing to understand two different "feature namespaces" and their interactions. An example of this is naming, where a programmer may need to turn to the filesystem to manage names in a completely orthogonal way to how the NVM frameworks handles data references. We will discuss another example, security, in our case study (§ 4).

Additionally, models that layer NVM programming atop existing interfaces often fail to facilitate effective persistent data sharing and protection. PMDK, an NVM programming library, makes design choices that limit scalability, since its data objects are not self-contained and do not have a large enough ID space, resulting in the need to coordinate object IDs across machines [10]. For the same reason, although single-address space OSes [12] somewhat address our first requirement, they do not consider both requirements at once, nor do they provide an effective and scalable solution to long-term data references due to that same coordination complexity [9].

2.2 A Data-Centric Approach

We cannot store virtual addresses in persistent data, so we need a new way to name a word of persistent memory: a *persistent pointer*. The persistent pointer encodes a persistent identification of data (§ 3.3) instead of an ephemeral address, allowing any thread to access the desired word of memory regardless of address space. This approach dramatically improves programmability, as programmers need not worry about the complexity of referring to persistent data with ephemeral constructs, improving data sharing across programs and runs of a program. Twizzler still makes use of virtual memory *hardware* to provide isolation and translation, but persistent data structures should not be written in terms of virtual addresses.

The Death of the Process. Processes as a first class OS abstraction are, like virtual addresses, unnecessary; a traditional process couples threads of control to a virtual address space, a security role, and kernel state. However, with the kernel removed from persistent data access, much of that kernel state (*e.g.* file descriptors) is unnecessary, leading to a decoupling of mechanisms: nothing fundamentally connects a virtual address space (*how* threads access data) and a security context (*what* data they may access). Instead, a data-centric OS can replace the process abstraction with security contexts, allowing greater flexibility for how security policy is managed.

The process abstraction is just one example. Persistent data access plays a key role in OS abstraction design, and we need to avoid complexity arising from combining old and new interfaces. Hence, we need to consider the wide-reaching effects of changing the persistence model on *all* aspects of the system, not just I/O interfaces. NVM gives us an opportunity to design an OS around the requirements of the target programming model instead of trying to mold support libraries around existing interfaces. While it is important that we provide support for legacy applications, it is these applications that should be relegated to support libraries; new applications built for the programming model should get first-class OS support.

Targeting these Constraints with Twizzler. The consequences of meeting the requirements of these hardware trends define a bounded design space for data-centric OSes. We have chosen a point in that space and built Twizzler, our approach to providing applications with efficient and effective access to NVM. In the following section we will discuss how our four primary abstractions—a low level persistent object model, a persistent pointer design, an address space mechanism called *views*, and a security context mechanism—achieve these goals of removing the kernel from the persistent data access path.

3 The Design of Twizzler

Twizzler is a stand-alone kernel and userspace runtime that provides execution support for programs. It provides, as first-class abstractions, a notion of threads, address spaces, persistent objects, and security contexts. A program typically executes as a number of threads in a single address space (providing backwards compatibility with existing programming models), into which persistent objects are mapped on-demand. Instead of providing a process abstraction, Twizzler provides *views* (§ 3.2) of the object space, which enable a program to map objects for access, and *security contexts* (§ 3.4) which define a thread's access rights to objects in the system. Twizzler provides persistent pointers (§ 3.3) for programs, as well as primitives to ensure crash-consistency (§ 3.5). The thread abstraction is similar to modern OSes; the kernel provides scheduling, synchronization, and management primitives. Figure 1 shows an overview of the system organization and how different parts of the system operate on data objects.

Twizzler's kernel acts much like an Exokernel [28,41], providing sufficient services for a userspace library OS, called `libtwz`, to provide an execution environment for applications. The primary job of `libtwz` is to manage mappings of persistent objects into the address space (§ 3.2) and deal with persistent pointers (§ 3.3). Twizzler also exposes a standard library that provides higher level interfaces beyond raw access to memory. For example, software that better fits message-passing semantics can use library routines that implement message-passing atop shared memory. Twizzler's standard library provides additional higher level interfaces, including streams, logging, event notification, and many others. Applications use these to easily build composable tools and pipelines for operating on in-memory data structures without the performance loss and complexity of explicit I/O.

We provide POSIX support with `twix`, a library that emulates Linux syscalls. We modified `musl` [1], a C library which all programs link to, replacing invocations of the `syscall` instruction with calls into `twix`, which internally tracks UNIX state like file descriptors. This is handled entirely in userspace; calls to `read` and `write` often reduce to calls to `memcpy`.

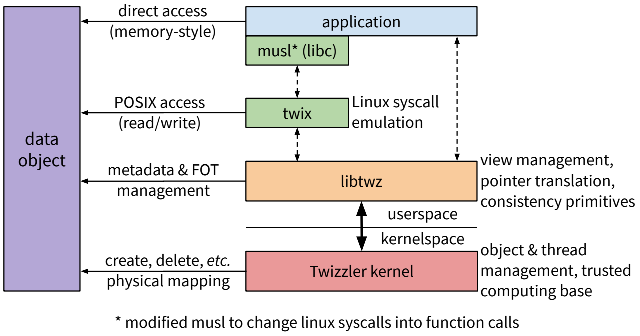

data object
direct access (memory-style)
application
musl* (libc)
POSIX access (read/write)
twix
Linux syscall emulation
metadata & FOT management
libtwz
view management, pointer translation, consistency primitives
userspace
kernelspace
create, delete, etc. physical mapping
Twizzler kernel
object & thread management, trusted computing base
* modified musl to change linux syscalls into function calls

Figure 1: Twizzler system overview. Applications link to `musl` (a C library), `twix` (our Linux syscall emulation library), and `libtwz` (our standard library).

3.1 Object Management

Twizzler organizes data into *objects*, which may be persistent. Each object is identified by a unique 128 bit object ID (though larger IDs would be possible). Objects provide contiguous regions of memory that organize semantically related data with similar lifetime. Applications access objects via mapping services (discussed in the next section) by mapping each object into a contiguous range in the address space, though the address space itself may be densely or sparsely mapped. Objects can be anywhere from 4 KiB (the size of a page) to 1 GiB; the upper bound on object size is a prototype implementation choice, and not fundamental to the design.

Twizzler uses objects as the unit of access control, building off a read/write/execute permissions model which mirrors that of memory management units in modern processors. This is a direct consequence of avoiding the kernel for persistent data access—it can set policy by programming the MMU, but must leave enforcement up to the hardware which, in-turn, defines what protections are possible.

An object, from the programmer's perspective, is flexible in its contents—for example, it could contain anywhere from a single B-tree node to the entire B-tree. Often, an object would contain the entire tree, since the entire tree is typically subject to the same access semantics by programs, and there are overheads associated with objects that can be amortized over larger spaces. Data and data structures that are too large for one object or require different access permissions can span multiple objects with references between them. We demonstrate the benefits of this flexibility in Section 4.

The kernel provides services for object management, such as creating and deleting objects. Objects are created by the `create` system call, which returns an object ID. A program may also optionally provide an existing object ID to the `create` call, stating that the new object should be a copy of the existing one, for which Twizzler uses copy-on-write. The new ID is a number that is unlikely to collide with existing IDs in the 128 bit ID space, and can be assigned using a technique that supports this requirement (random, hashing, *etc*.). Some forms of ID assignment support a form of access control: a program can only access an object whose ID it knows. Twizzler provides object naming as well, discussed in Section 3.3.

Objects may be be deleted via the `delete` system call. Like Unix's `unlink`, objects are reference counted, where a reference refers to a mapping in an address space. Once the reference count reaches zero, the object may be deleted.

3.2 Address Space Management

Although virtual addresses are the wrong abstraction to use for persistent data access, we do leverage virtual address hardware in modern processors for isolation and protection. Twizzler provides access to persistent objects by mapping them into the virtual address space behind-the-scenes (via `libtwz`). This generates many mapping operations to access persistent data, so requiring system calls would be costly. Additionally, our kernel avoidance necessitates an increased address space management responsibility for userspace. For example, executable loading and mapping is handled largely without the kernel.

To support userspace manipulation of address spaces, the kernel and userspace share an object (called a "view") that defines an address space layout. The view is just a normal object, and so standard access control mechanisms apply to enforce isolation. When applications map objects into their address space, they update the view to specify that a particular object should be addressable at a specific location. The kernel then reads the object and determines the requested layout of the virtual address space. The view object is laid out like a page-table, where each entry in the table corresponds to a slot in the virtual address space. Each table entry contains an object ID and read, write, and execute protection bits to further protect object access (like `PROT_*` in `mmap`).

When a page-fault occurs, the fault handler tries to handle the fault by either doing copy-on-write, checking permissions, or by trying to map an object into a slot if the view object requested one. If it cannot handle the fault (due to a protection error or an empty entry in the view object), it elevates the fault to userspace where `libtwz` handles it, possibly by killing the thread, or possibly by mapping an object if the slot is "on-demand". When the kernel maps an object into a slot, it updates the address space's page-tables appropriately.

When threads add entries to a view object they need not inform the kernel—when a fault occurs, the kernel will read the entry as needed. However, when *changing* or *deleting* an entry, threads must inform the kernel so it can update existing page table entries. We provide two system calls for views. The `set_view` call allows a thread to change to a new view, which might be used to execute a new program or jump across programs to, for example, accomplish a protected task. Twizzler's access control system prevents this from happening arbitrarily. The second system call is `invalidate_view`, which lets a thread inform the kernel of changed or deleted entries.

3.3 Persistent Pointers

Section 2 discussed the needs for references that outlive ephemeral actors. Twizzler provides *cross-object* persistent pointers so that a pointer refers not to a virtual address but to an offset within an object by encoding an `object-id:offset` tuple. This enables a pointer to refer to persistent data, but it also allows objects to have *external* pointers that refer to data in any object in the global object space. We highlight cross-object pointers' power and flexibility by demonstrating their ability to express inter-object relationships in Section 4.

To efficiently encode this tuple, we use indirection through a per-object *foreign object table* (FOT), located at a known offset within each object. The FOT is an array of entries that each stores an object ID (or a name that resolves into an object

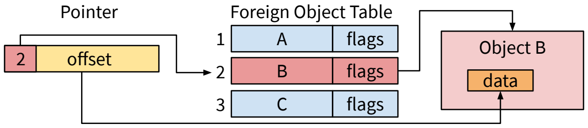


Figure 2: Pointer translation via the FOT. The pointer and the FOT are both contained in the same object (not shown).

ID, as we will see below) and flags. A cross-object pointer is stored as a 64 bit `FOT_idx:offset` value, where the `FOT_idx` is an index into the FOT. This provides us with both large offsets *and* large object IDs, since the IDs are not stored within the pointer itself. If an object wishes to point to data within itself (an *intra-object* pointer), it stores 0 in `FOT_idx`. When dereferencing, Twizzler uses the `FOT_idx` part of the pointer as an index into the FOT, retrieving an object ID. The combination of a FOT and a cross-object pointer logically forms an `object-id:offset` pair, as shown in Figure 2.

Our design (discussed in prior work [9, 10]) differs from existing frameworks [6, 13, 18, 19, 57, 58] because of the indirection. Frameworks like PMDK store entire object IDs within pointers, increasing pointer size and reducing flexibility by removing the possibility of late-binding (discussed below). Additionally, Twizzler extends the namespace of data objects beyond one machine, as machine-independent data references are a natural consequence of cross-object pointers. Existing solutions are limited in this scalability. They either limit the ID space (necessary for storing IDs in pointers) and thus resort to complex coordination or serialization when sharing, or they require additional state (*e.g.* per-process or per-machine ID tables) that must be shared along with the data, forcing the receiving machine to "fix-up" references. Worse still, the fix-up is application-specific, since the object IDs are within any pointer, not in a generically known location. Our per-object FOT results in self-contained objects that are easier to share, thus interacting better with remote shared memory systems.

Part of our motivation for this FOT indirection was to allow a large ID space without increasing pointer size. PMDK, by contrast, increases pointer size to 128 bits for each pointer. Twizzler has no additional space overhead per-pointer, instead adding a 32-byte overhead per FOT entry. The number of FOT entries, however, is typically much smaller than the number of pointers since pointers to the same external object can all use the same FOT entry. As we will see in Section 5, this has a dramatic benefit to performance.

FOT Entries and Late-Binding. The FOT entry's `flags` field has bits for read, write, and execute protections. The protections are *requests*; Twizzler implements separate access control on objects. This allows some pointers to refer to data with a read-only reference while others can be used for writing, reducing stray writes (a single ID can repeat in the FOT with different protections). The FOT entries also enable atomic updates that apply to all pointers using that FOT entry.

Instead of *requiring* programmers to refer to objects via IDs only, we allow names in FOT entries. These entries may contain a pointer to an in-object string table that contains a name. Names enable late-binding [19], a vital aspect of systems, allowing references to objects which change over time, *e.g.* shared library versions. Names are passed to a *resolving* function (specified in the FOT entry). Allowing a program to specify how its names are resolved increases the flexibility of the system beyond supporting UNIX paths. Twizzler provides a default name resolver that uses UNIX-like paths.

The implementation of naming is orthogonal to Twizzler's design. We allow a range of name resolution methods within the system stack and allow objects to specify their own name resolution functions for flexibility. For example, objects could be organized by both a relational database and a hierarchical namer similar to conventional file systems. Non-hierarchical file systems are well studied [3,31,32,54,55], but these systems do not easily cooperate atop a single data space. Since Twizzler uses a flat namespace as its "native" object naming scheme, it enables the required cooperation.

Pointer Translation. Current processors provide only a virtual memory abstraction, so applications must do some extra work to dereference a pointer, *translating* a pointer from its persistent form into a virtual address. This does not affect the *stored* pointer, which is still persistent and independent of any translation or address space. Thus multiple applications, possibly with different address space layouts, can translate the same pointer at the same time without coordination.

Pointer translation occurs with the help of two `libtwz` functions: `ptr_lea` (load effective address) and `ptr_store`. When a program dereferences a pointer, it first calls `ptr_lea`. The pointer is resolved into an object-ID and offset pair through a lookup in the FOT, after which `libtwz` determines if the referenced object is already mapped (by maintaining per-view metadata). If not, it picks an empty slot in the view and maps the object there (a cheap operation that does not invoke the kernel). Once mapped, `libtwz` combines the object's temporary virtual base address with the offset, and returns the new pointer. The `ptr_store` function does the opposite of `ptr_lea`—it turns a virtual pointer into a persistent one. While these are done manually in our implementation, we plan to implement compiler support to emit these calls automatically.

FOT management is handled by `libtwz`. While a lookup in the FOT is a simple array-indexing operation, a store may require adding to the FOT. To avoid duplicate entries, `libtwz` walks the FOT looking for a compatible entry. If one is not found, it atomically reserves a new entry and fills it (flushing cache-lines to persist it) before storing the pointer. The `ptr_store` operation is less common than `ptr_load`, and in the future we may include additional caching metadata that would speed-up the FOT walk (such as storing recent IDs).

Translating pointers has a small overhead (§ 5) and the result can be cached. Twizzler improves performance via a per-object cache of prior translations. The common case, intra-object pointers, does not require an external lookup and is implemented as a simple bitwise-or operation.

3.4 Security and Access Control

Twizzler's focus on memory-based objects requires that we design the security model around hardware-based enforcement, where the MMU checks each access. This design is *inevitable* in a data-centric OS, since the kernel is not involved in every memory access. The kernel merely specifies the access rights when mapping an object and then relies on the hardware to enforce those rights with a low overhead.

A key design choice we make is *late-binding on security*. Applications request access to an object with permissions that they desire; if they access the object in only allowed ways (*e.g.*, only reading a read-only object), no fault occurs. This is because when we map an object (via a view), the kernel is not immediately involved, and so cannot check access rights for a particular access at the time the mapping is setup. Performing an access rights check on time of first access does not make sense either, as it associates a specific access (that might be allowed) with a permissions error. For example, if a program reads object A, and that program is allowed to read A, it should be allowed to perform the read even if it requested read-write access to the object. This late-binding enables simpler programs that need not worry about elevating access rights through remapping data objects. Programs can make progress without knowing in advance the permissions of the objects they might access, thus enabling the reuse of the OS's access control mechanism in applications. We will show the flexibility of this in Section 4, wherein we add access control to a program by changing only a few lines of code.

Threads run in a security context [8, 25, 44], which contains a list of access rights for objects and allows the kernel to determine the access rights of programs. Using these contexts, Twizzler is able to provide analogues to groups and owners in Unix while providing more fine-grained access control if necessary. Unlike past exploration into security contexts, data-centric OSes offer an advantage in simplicity. A security context abstraction in a Unix-like OS needs to maintain access rights to a set of fundamentally different things (such as paths, virtual memory locations, and system calls). Instead, Twizzler's security contexts specify access rights to an object via IDs instead of virtual addresses. This also makes security contexts persistent, allowing us to use them as the primary way we assign security roles to threads.

Security contexts are implemented via virtualization hardware that maps virtual memory to an intermediate "object space" which specifies the access rights, which is then mapped to physical memory [9]. This reduces the number of page-table structures and mappings, as threads in the same security context can share the same page-tables for each object.

3.5 Crash Consistency

Twizzler provides primitives for building crash-consistent data structures. At a low level, it provides a mechanism for writing back cache-lines and appropriate fences. Applications use these primitives today outside of Twizzler to build up larger, more complex support for crash-consistent data structures.

Our goal is to provide low level primitives without restricting programs or prematurely prescribing particular solutions. There is a wealth of research on crash-consistent data structures for NVM [15, 16, 24, 46, 50–53, 65], but it is still in flux. Of course, Twizzler manages *system* data structures, such as FOT entries, views, *etc.*, in a crash-consistent manner using the aforementioned primitives, locking, and fencing.

Twizzler also provides a transactional-persistent logging mechanism. Programmers can write `TXSTART`–`TXEND` blocks to denote transactions and `TXRECORD` statements to record pre-changed values. This is similar to the mechanism provided by PMDK [58]. If applications need more complex transactions using different logging mechanisms, they can use libraries.

Twizzler provides a mechanism for restarting threads when power is restored following a crash. Since views are persistent objects, all mapped objects during a thread's execution are known across power cycles, and are mapped back in. The thread is then started at a special `_resume` entry point, allowing the program to handle the power failure in an application-specific manner with access to the state of the program (data segment, heap, *etc.*) as it was when power was cut.

3.6 Implementation

Twizzler's kernel is similar to many microkernels, providing a small set of key primitives. It is 5,500 lines of architecture-independent code and 5,700 lines of architecture-dependent CPU driver code. The primary complexity in the system is implemented in userspace, as the design of the programming model greatly simplifies the kernel. Twizzler is open-source; more information can be found at https://twizzler.io.

We also built a prototype of Twizzler by modifying the FreeBSD 11.0 kernel before implementing our standalone kernel. This was done both to more rapidly verify our design and to provide a prototyping environment for developers to write code for Twizzler in a familiar environment. We added Twizzler services to FreeBSD by adding system calls, modifying the fault-handling logic, and distinguishing Twizzler threads from FreeBSD threads. This is also a testament to the simplicity of the kernel in our model, since FreeBSD was relatively easy to modify to support the Twizzler userspace. However, the FreeBSD prototype is limited by its need to coordinate with FreeBSD's Unix services, thus the standalone kernel is more efficient and simpler, and provides a better environment for researching kernel design changes in the face of NVM.

4 Evaluation

Our primary goals for evaluating Twizzler were:

1. Show that Twizzler meets the needs of a data-centric OS in enabling programs to directly access persistent data.

2. Demonstrate that the programming model we defined provides sufficient power to easily and effectively build real applications with NVM in mind.
3. Measure the performance of our system to understand where we gain and lose performance.

We approached these goals two ways: porting existing software (SQLite) and writing new software for Twizzler. The first demonstrates both the generality of the programming environment (legacy software can be easily ported) and the potential performance gains to be had even for legacy software. The second demonstrates the true power of Twizzler's programming model and allows us to explore the consequences of our design choices fully without being constrained by legacy designs.

We built three pieces of new software: a hash-table based key-value store (KVS), a red-black tree data structure, and a logging daemon. Each had different characteristics and goals, and together they demonstrate the flexibility that Twizzler offers in allowing simple implementation, nearly-free access control, and the ability to directly express complex relationships between objects. Using our KVS and red-black tree code, we ported SQLite (a widely used SQL implementation) to Twizzler along with a YCSB [17,29] driver (a common benchmark), allowing us to explore Twizzler's model in a larger, existing program that would let us study the performance of Twizzler in a complex system that stores *and processes* data. We present the performance of SQLite and our new software, along with microbenchmarks, in Section 5.

4.1 Case Study: Key-Value Store

We implemented a multi-threaded hash-table based key-value store (KVS), called twzkv, to study cross-object pointers and our late-binding of access control. Our KVS supports insert, lookup, and delete of values by key (both of arbitrary size), and hands out direct pointers to persistent data during lookup. During insert, it copies data into a data region before indexing the inserted key and value. We built twzkv in multiple phases to study how our system handles changing requirements.

We built twzkv in roughly 250 lines of C. Handing out direct pointers into data was trivial to implement with cross-object pointers, requiring only a call to ptr_lea during lookup. The initial implementation maintains two objects, one for data and one for the index. The complexity typically involved when storing both index and data in a single, flat file is not justified in a programming model where we can express inter-object relationships directly at near-zero cost in complexity or performance. In our case, a pointer from the index object to the data object (such as an entry in the hash table) can be written with a single call to ptr_store. This, combined with the simple requirements for an in-memory NVM KVS, resulted in a small implementation that was nonetheless a usable KVS.

Extending Requirements. Next, we added functionality to protect values with access control. We wanted to keep handing out direct pointers to data during lookup and to keep twzkv a library (as opposed to a service). Meeting these goals on an existing system would be difficult without adding significant complexity, such as reimplementing a lot of Twizzler's pointer framework or implementing manual, redundant access control.

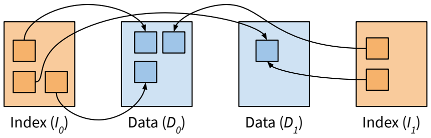


Figure 3: Cross-object pointers in twzkv.

In Twizzler, implementing access control in twzkv involved having the index refer to data in multiple data objects, assigning those objects different access rights, and allocating from those objects depending on desired access rights. We were able to implement this while preserving the original code due to the transparent nature of Twizzler's cross-object pointers. Now, when inserting, the application indicates the data object into which to copy the data, as shown in Figure 3.

By supporting multiple data objects, twzkv can leverage the OS's access control, sidestepping complexity. Unrestricted data can go in D_0 (Figure 3), whereas restricted data can go in D_1. Since each object has distinct access control, a user can set the objects' access rights, then decide where to insert data according to policy. The indexes point to the correct locations regardless of the access restrictions of the data objects, and twzkv still hands out direct pointers, but a user that is restricted from accessing data in D_1 will not be able to dereference the pointer. A further extension is to support secondary indices, as shown in Figure 3, enabling alternative lookup methods and limiting data discovery with index object access control. This extension is easy to implement on Twizzler.

Comparison to Unix Implementation. To compare with existing techniques, we built a similar KVS using only Unix features (called unixkv). It also separates index and data, but it must manually compute and construct pointers. Supporting multiple data objects was complex in unixkv, because we had to store and process file paths in the index and store references to paths for pointers, increasing overhead and code complexity by 36%—a lot for an implementation with relatively few pointers—just to reimplement Twizzler's support. The extra complexity also included code to manually open, map, and grow files, much of which Twizzler handles internally. Development time was extended by bugs that were not present when developing twzkv, due to the manual pointer processing. While twzkv gains transparent access control, unixkv does not due to the lack of on-demand object mapping and late-binding of security. Instead, unixkv needs to know object permissions before mapping, a restriction that limits the ability to reuse OS access control, something that twzkv could leverage through late-binding on security (§ 3.4)[1]. Other frameworks like PMDK that do not integrate access control and late-binding into their models have similar limitations.

[1] unixkv could trap segmentation faults to do this, but that would be application-specific, difficult, and would reimplement Twizzler functionality.

4.2 Case Study: Red-Black Tree

To evaluate the process of writing persistent, "pointer-heavy" data structures, we implemented a red-black tree in C using normal pointers (ramrbt) in 100 lines of code, and evolved it for persistent memory in two ways: manually writing base+offset style pointers, as current systems require (unixrbt), and using Twizzler (twzrbt). Porting existing data structure code to persistent memory will be common during the adoption of NVM, and much of the complexity therein comes from dealing with persisting virtual addresses [47].

In developing unixrbt, we found 83 locations where we had to perform pointer arithmetic for converting between persistent and virtual addresses. Consider an expression such as `root->left->right = foo`. Inserting calls to translate this directly results in `L(L(root)->left)->right = C(foo)`, where `L` converts to a virtual address and `C` converts back, which is heavily obfuscated and took more development time than writing ramrbt in the first place due to debugging.

We built twzrbt like unixrbt, annotating pointer stores and dereferences. However, unixrbt used an application-specific solution for pointer management; if other applications wanted to use the data structures created by unixrbt, they would have to know the implementation details of the pointer system (or share the implementation, thus reimplementing much of Twizzler's library). Additionally, due to Twizzler enabling improved system-wide support for cross-object pointers, these transformations can be made automatic.

Unlike twzrbt, unixrbt's tree is limited to a single persistent object; a limitation that prevents the tree from growing arbitrarily, does not allow it to directly encode references to data outside the tree object, and does not gain it the benefits of cross-object data references that were discussed above for twzkv. Adding support for this to unixrbt would require modifying the core data structures to include paths and significantly altering the code, increasing its length by at least a factor of 2, whereas twzrbt gets this functionality for free.

Another advantage of twzrbt is reduced support code compared to unixrbt; unixrbt needed code to manage and grow files and mappings, while we implemented twzrbt as simple data structure code with Twizzler managing that complexity. The additional error handling code and pointer validity checks in unixrbt (handled automatically in Twizzler) increased development time and implementation complexity.

4.3 Porting SQLite

We ported SQLite to Twizzler to demonstrate our support for existing software and to evaluate the performance of a SQLite backend designed for Twizzler. We used our POSIX support framework, a combination of musl and our library twix, to support much of SQLite's POSIX use. We took a modified version of SQLite called SQLightning that replaced SQLite's storage backend with a memory-mapped KVS called LMDB [14]. We chose this port because LMDB is implemented with mmap'd files as the primary access method and hands out direct pointers to data as one would expect from an effectively designed NVM KVS[2]. Since LMDB's SQLightning port already replaces the storage backend with calls to LMDB, we ported SQLite to Twizzler by taking our KVS and red-black tree code and implementing enough of the LMDB interface for SQLite to run using Twizzler as a backend. Outside of the B-tree source file few changes were needed for SQLite to run on Twizzler. We further ported our modified SQLite backend to PMDK to compare directly with a commonly used NVM programming library that supports persistent pointers.

We also ported a C++ YCSB driver [29], which required porting the C++ standard template library (STL). Since we had already ported a standard C library, the C++ STL was easily ported, demonstrating the ease of porting software to Twizzler. We have also ported some existing UNIX utilities (such as bash and busybox), which largely require only recompiling to run on Twizzler. Of course, to gain *all* of the benefits of Twizzler, programs will be need to be written with NVM in mind (but this is true regardless of the target OS).

Our implementation of the LMDB interface corroborated our experience from the KVS case study: much of the complexity in storage interfaces and implementations comes from the separation between storage and memory. This has been studied before (as we will elucidate in Section 6), but the advent of NVM changes the game significantly by allowing programmers to think directly via in-memory data structures. The result is that interfaces like cursors in a KVS become redundant. We implemented to this interface for LMDB, but the functions were largely wrappers around storing a pointer to a B-tree node and traversing the tree directly without separate loads and copies. The result was an extremely simple implementation (500 LoC) that still met the required interface. Future software for NVM can use Twizzler's programming model to more effectively write software that eschews the need for complexity forced by the two-tier storage hierarchy.

4.4 Discussion

Although these implementations were simple, they represent the applications and data structures we expect in a data-centric system. Pointers we can directly use in our programming languages make computing over persistent data almost transparent, allowing simple implementations that are nevertheless easy to evolve as requirements change.

Not only does twzkv have access control, but it enables concurrent access via cross-object pointers. Applications can load indexes for multiple databases without needing to worry about address space layout and without writing complex pointer management code that would be required by an implementation using mmap. We were able to provide access control without a single line of code in twzkv dedicated to checking

[2] These are not persistent pointers, however, unlike Twizzler's.

or enforcing access rights. Instead, we relied on the system's access control, something not possible with other frameworks that do not support late-binding of access rights and do not consider security as part of their programming model. Twizzler thus removes the need for applications to manage their own access control, which increases the security of the system by divesting programmers from the responsibility of getting it right. Similar functionality for current systems would traditionally require separation of the library and application into a client-server model, but that additional overhead is unneeded here and inappropriate on a persistent memory system.

Although `twzrbt` and `twzkv` had different densities of pointer operations, `twzrbt` being "pointer-heavy" and `twzkv` being "pointer-light", Twizzler improved the complexity of both over manual implementation and improved flexibility over existing persistent pointer methods. Using a system-wide standardized approach to pointer translations not only enables better compiler and hardware support, but it also improves interoperability; because they share a common framework, `twzkv` could use the red-black tree code and data with ease, and even interact with the SQLite database even though they were written separately without that goal in mind. The position-independence afforded by this model enables both composability and concurrency, while also simplifying programming on persistent data to a natural expression of data structures.

Non-Shared-Memory Programs. To push the limits of our model and show that Twizzler does not constrain programmers into a shared-memory model, we implemented a logging framework (similar to `syslogd`). The logging daemon, `logboi`, can receive log messages either synchronously or asynchronously. In both cases, the interface is the same, but synchronous logging uses shared-memory abstractions while asynchronous logging relies on message-passing semantics.

For synchronous logging the thread switches security contexts, which is made possible by decoupling address spaces and security. The call to the logging framework then updates the log and returns. An asynchronous logging event sends data to the logging thread via a stream object (a standard API provided by Twizzler) that `logboi` and the application share. The choice of asynchronous or synchronous is left to the programmer; synchronous can have lower latency and predictable behavior while asynchronous offloads processing to `logboi`.

5 Performance

Our evaluation's primary focus is on the benefits of the programming model, showing new functionality with reduced complexity at an acceptable overhead. Nevertheless, there are many cases where we see significant improvement (such as SQLite) because the programming model has less overhead, and our pointer design is space efficient and fast to translate.

We measured the performance of our KVS and red-black tree, performed microbenchmarks, and evaluated the Twizzler port of SQLite against Linux (Ubuntu 19.10) instances of SQLite, SQLightning, and our port of SQLite to PMDK. Tests ran on an Intel Xeon Gold 5218 CPU running at 2.30 GHz with 192 GB of DRAM and 128 GB of Intel Persistent DIMMs. We compiled all tests against the `musl` C library instead of `glibc` because Twizzler uses `musl` to support Unix programs.

All Linux tests used the NOVA filesystem [69] (a filesystem optimized for NVM) on the NVDIMMs, mounted in DAX mode. This enabled direct access to the persistent memory without a page-cache interposing on accesses.

Table 1: Latency of common Twizzler operations.

Pointer Resolution Action	Average Latency (ns)
Uncached FOT translation	27.9 ± 0.1
Cached FOT translation	3.2 ± 0.1
Intra-object translation	0.4 ± 0.1
Mapping object overhead	49.4 ± 0.2

5.1 Microbenchmarks

Table 1 shows common Twizzler functions' latencies, including pointer translation. The overhead shown for resolving pointers does not include dereferencing the final result, since that is required regardless of how a pointer is resolved. The first row shows the latency for resolving pointers to objects the first time. Twizzler makes a further optimization by caching the results of translations for a given FOT entry. Each successive time that FOT entry is used to resolve a pointer, the result of the original translation is returned immediately, improving the latency as shown on the "cached" row of Table 1. Note that the low latency of these results is expected; the performance critical case of these functions' use is repeated calls, and since these operations are simple, they fit within the processor cache.

Twizzler translates intra-object pointers by first checking if the pointer is internal and, if so, adding the object's base address to it—the same operation required for application-specific persistent pointers. The expanded programming model offered by Twizzler makes this overhead minor relative to the high costs for persistent data access on current systems, which have high-latency for equivalent operations.

We compared our pointer translation to Unix functions. Resolving an external pointer with an ID corresponds roughly to a call to `open("id")`, which has a latency of 1036 ± 15 ns. The comparison is not exact, of course; the pointer resolution also maps objects, and the call to `open` must handle file system semantics. However, the direct-access nature of NVM results in pointer translation achieving the same goal as opening a file does today. The pointer operations in Twizzler accomplish much of the same functionality as the heavier-weight I/O system calls on Unix with more utility and less overhead.

A more direct comparison is object mapping, which has low latency compared to `mmap` (658.7 ± 12.7 ns—a 13.3× speedup) though the two have similar functionality. Since map-

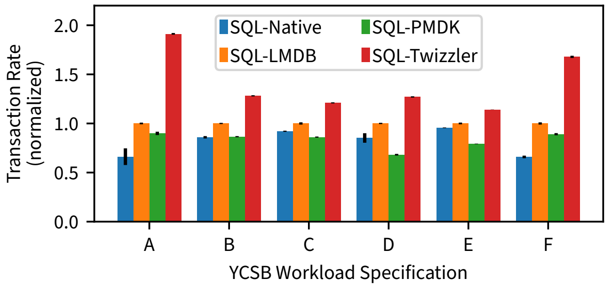


Figure 4: YCSB throughput, normalized (higher is better).

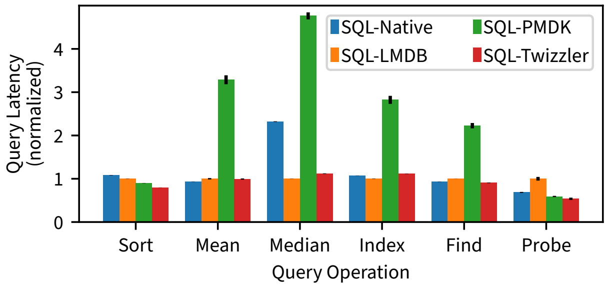


Figure 5: Query latency, normalized (lower is better).

ping occurs entirely in userspace, cache pollution is reduced. While both `mmap` and Twizzler's mapping require page-faults to occur before the data is actually mapped, this overhead is similar in Twizzler and Unix, and so is not shown.

5.2 SQLite

We ran four variants of SQLite, three on Linux and one on Twizzler, and compared their performance: "SQL-Native" (unmodified SQLite), "SQL-LMDB" (SQLite using LMDB as the storage backend), "SQL-PMDK" (SQLite using our red-black tree on PMDK), and "SQL-Twizzler" (our port of SQLite running on Twizzler). SQL-Native was run in `mmap` mode so that both it and SQL-LMDB used `mmap` to access data. We ran each on the same hardware and normalized the results.

Figure 4 shows the three variants' throughput under standard YCSB workloads. The performance improvement of the LMDB and Twizzler variants over SQL-Native is likely due to handing SQLite direct pointers to data. However, in the Twizzler case we get an additional benefit of operating on data structures directly while LMDB has an abstraction cost.

Figure 5 shows the latency of queries on a one million row table. This is common data processing—loading and then examining data in a variety of ways. We measured the performance of calculating the mean and median, sorting rows, finding a specific row, building an index, and probing the index. SQL-Twizzler had similar performance to SQL-LMDB and SQL-Native despite comparing its extremely simple storage backend to optimized B-tree backends (that benefit from scan operations). As a more direct comparison, SQL-Twizzler significantly out-performed SQL-PMDK in most tests. PMDK's pointer operations are more expensive than Twizzler's, requiring up to two hash table lookups per translation [5]. Additionally, PMDK's pointers are 128 bits, while Twizzler does not increase pointer size. Increased pointer size results in significantly worse cache performance, especially in a pointer-heavy data structure like a persistent red-black tree.

5.3 Key Value Store

We compared `twzkv` to `unixkv` by inserting one million distinct key-value pairs, followed by looking up each in-order.

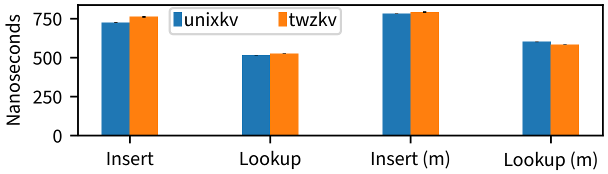


Figure 6: Latency of insert and lookup in `twzkv` and `unixkv`. An "(m)" indicates support for multiple data objects.

The inserted items were 32-bit keys and 32-bit values, chosen to reduce the overhead of data copying since we were focusing on pointer translation overhead. Both were compared under two modes, single-data-object and multiple-data-objects. Both KVSes translated between virtual and persistent addresses when storing and retrieving data, but for multiple-data-objects, we allow for storing the data in an arbitrary object.

Figure 6 shows the latency of lookup and insert, demonstrating that not only is the memory-based index and data object structure that can hand out direct data pointers sufficiently low latency to take advantage of NVM, but the additional overhead of cross-object pointers is minimal. Compared to `unixkv`, `twzkv` has minimal overhead in the single-object case, and improves lookup performance in the multiple-object case. The minor overhead in other cases comes with improved flexibility, simplicity, and access control support (`unixkv` does not support access control). Finally, multithreaded access on `twzkv` and `unixkv` did not improve performance; despite the pointer translations, they ran at memory bandwidth (for NVM).

5.4 Red-Black Tree

We measured the latency of insert and lookup of 1 million 32-bit integers on both `unixrbt` and `twzrbt`. The insert and lookup latency of `twzrbt` was 528 ± 3 ns and 251.8 ± 0.5 ns, while insert and lookup latency of `unixrbt` was 515 ± 2 ns and 213 ± 1 ns. The modest overhead comes with significantly improved flexibility, as `unixrbt` does not support cross-object trees, and less support code (`unixrbt` manually implements mapping and pointer translations). Note that even though there is lookup overhead in `twzrbt`, this overhead did not predict the results of a larger program—the SQL-Twizzler port used this red-black tree, and saw performance benefits over block-based implementations.

6 Related Work

Twizzler's design is shaped by fundamental OS research [12, 18, 26–28, 41, 42], which, while approaching similar topics described in Section 2, often did not consider *both* design requirements simultaneously, resulting in an incomplete picture for NVM. Recent research on building NVM data structures [15, 16, 22, 37, 45, 65], often focuses on building data structures that provide failure atomicity and consistency. In contrast, we explore how NVM affects programming models. We draw from recent work on providing OS support for NVM systems [11] and work providing recommendations for NVM systems [48], integrating object-oriented techniques and simplified kernel design to provide high-performance OS support for applications running on a single-level store [4, 61].

Multics was one of the first systems to use segments to partition memory and support relocation [6, 19]. It used segments to support location independence, but still stored them in a file system, requiring manual linkage rather than the automated linkage in Twizzler. Nonetheless, Multics demonstrated that the use of segmenting for memory management can be a viable approach, though its symbolic addresses were slow.

The core of Twizzler's object space design uses concepts from Opal [12], which used a single virtual address space for all processes on a system, making it easier to share data between programs. However, Opal was a single-address space OS, which is insufficient for NVM [9, 10], and it did not address issues of file storage and name resolution. It also required a file system, since there was no way to have a pointer refer to an object with changing identity, whereas our approach removes the need for an explicit file system. Other single-address space OSes, such as Mungi [34], Nemesis [56], and Sombrero [63], show that single address spaces have merit, but, like Opal, did not consider how the use of NVM would alter their design choices; in particular, how the use of fixed addresses results in a great deal of coordination that is unnecessary in our approach. OSes such as HYDRA [68] provide functionality similar to cross-object pointers; however, in Twizzler, we extend their use from procedures-referencing-data to a more general approach. Furthermore, they required heavy kernel involvement, an approach incompatible with our design goals.

Single-level stores [21, 60, 62] remove the memory versus persistent storage distinction, using a single model for data at all levels. While well-known, "little has appeared about them in the public literature" [60], even since the EROS paper. Our work is partially inspired by Grasshopper [21], AS/400, and orthogonal persistence systems, but while these are designed to provide an illusion of persistent memory, Twizzler is built for real NVM and focuses on providing a truly global object space with global references without cross-machine coordination. Clouds [20] implemented a distributed object store in which objects contained code, persistent data, and both volatile and persistent heaps. Our approach uses lighter-weight objects, allowing direct access to objects from outside, unlike Clouds. Software persistent memory [33], designed to operate within the constraints of existing systems, built a persistent pointer system using explicit serialization without cross-object references, in contrast to Twizzler. Meza [49] suggested hardware manage a hybrid persistent-volatile store with fine-grained movement to and from persistent storage. Since persistence in Twizzler is to NVM, we need not interpose on movement between storage and memory, instead simply managing memory mappings of persistent objects, reducing OS overhead.

Recently, several projects have considered the impact of non-volatile memories on OS structure. Bailey, *et al.* [4] suggest a single-level store design. Faraboschi, *et al.* [30] discuss challenges and inevitable system organization arising from large NVM, and we follow many of their recommendations. The Moneta project [11] noted that removing the heavyweight OS stack dramatically improved performance. While Moneta focused on I/O performance, not on rethinking the system stack, we leverage their approach to reduce OS overhead as much as possible, even when the OS must intervene. Lee and Won [43] considered the impact of NVM on system initialization by addressing the issue of system boot as a way to restore the system to a known state; we may need to include similar techniques to address the problem of system corruption.

IBM's K42 [42] inspired the high level design of Twizzler. The object-oriented approach to designing a micro or exokernel used in K42 is an efficient design for implementing modular OS components. Like K42, Twizzler lazily maps in only the resources that an application *needs* to execute. Similar techniques for faulting-in objects at run-time have been studied [36]. Communication between objects in Twizzler is, in part, implemented as protected calls, similar to K42.

Emerald [39, 40] and Mesos [35] implemented networked object mobility, which we can also support. Emerald implemented a kernel, language, and compiler to allow objects mobility using wrapper data structures to track metadata and presenting objects in an object-oriented language, impacting performance via added indirection for even simple operations.

The Twizzler object model was shaped by NV-heaps [15], which provides memory-safe persistent objects suitable for NVM and describes safety pitfalls in providing direct access to NVM. While they have language primitives to enable persistent structures, Twizzler provides a lower-level and uninhibited view of objects like Mnemosyne [65], allowing more powerful programs to be built. Languages and libraries may impose further restrictions on NVM use, but Twizzler itself does not. Furthermore, Twizzler's cross-object pointers allow external data references by code, whereas NV-heap's and DSPM's [59] pointers are only internal. Existing work beyond Multics on external references shows and recommends hardware support [58, 66], but provides a static or per-process view of objects, unlike Twizzler, limiting scalability and flexibility.

Projects such as PMFS [24] and NOVA [69] provide a file system for NVM. Twizzler, in contrast, provides direct NVM access atop of a key-value interface of objects. Although Twizzler does not supply a file system, one can be built atop it. While NOVA and PMFS provide direct access to NVM, NOVA adds indirection with copies. Both use `mmap` (which falls short as discussed above) and, unlike Twizzler, require significant kernel interaction when using persistent memory.

Our kernel that "gets out of the way" is influenced by systems such as Exokernel [28] and SPIN [7], both of which drew on Mach [2]. In Exokernel, much of the OS is implemented in userspace, with the kernel providing only resource protection. Our approach is similar in some respects, but goes further in providing a single unified namespace for all objects, making it simpler to develop programs that can leverage NVM to make their state persistent. In contrast, SPIN used type-safe languages to provide protection and extensibility; our approach cannot rely upon language-provided type safety since we want to provide a general purpose platform.

7 Future Work

Compiler and Hardware Support. Clean-slate NVM abstraction reopens the possibility of coevolving OSes, compilers and languages, and hardware. Standardized OS support for cross-object pointers enables compiler support more effectively than application-specific solutions [47] or simple libraries [58]. Twizzler's pointer translation functions are simple enough to be automatically emitted by a compiler. Similarly, designing an OS for cross-object pointers allows us to better state our needs to hardware, which can alleviate performance overheads for pointer translation [66, 67].

Security. Although we discussed the Twizzler security model briefly, there is still much to do. The current model provides access control, a basic ability to define and assign roles based on security contexts, and simple sub-process fault isolation through the ability to switch security contexts. We are exploring a *flexible* security model that allows programmers to easily trade-off between security, transparency, and performance using capability-based verification. For example, we are implementing a call-gating mechanism that will allow us to restrict control-flow transfers between application components, improving the security against malicious components and reducing the possibility of memory-corrupting bugs.

Networking and Distributed Twizzler. One of the key principles of Twizzler is to focus the programming model on data and away from ephemeral actors such as processes and nodes. This is enabled by our identity-based references that decouple location from references, and by ensuring all the context necessary to understand these relationships is stored with the data. Because our data relationships are independent of the context of a particular machine, applications can more easily share data. This easy sharing, combined with a large ID address space, motivates a *truly* global object ID space.

We are building a networking stack and support for a distributed object space into Twizzler. Our networking stack is based around extensive use of hardware virtualization in modern NICs. This design, which is in use in existing kernel-bypass strategies, will mesh well with our core OS design of reducing kernel interposition. At a higher level, we are considering how distributed applications change in our model. For example, an increase in data mobility facilitated by our location-independent data references and identities means that we can manifest both data and code where they are needed without complex marshalling, turning distributed computation into a rendezvous problem. We plan to build distributed applications atop Twizzler to demonstrate this approach.

Of course, for compatibility we will provide a traditional sockets-based networking stack. However, we can use existing userspace libraries that, *e.g.*, implement TCP in userspace. Because we implemented our POSIX compatibility library in userspace, applications can gain many benefits afforded by kernel-bypass networking frameworks while still using traditional socket interfaces.

8 Conclusion

Operating systems must evolve to support future trends in memory hierarchy organization. Failing to evolve will relegate new technology to outdated access models, preventing it from reaching full potential, and making it difficult for OSes to evolve in the future. Twizzler shows a way forward: an OS designed around NVM that provides new, efficient, and easy to use semantics for direct access to memory. Cross-object pointers in Twizzler allow programmers to easily build composable and extensible applications with low overhead by removing the kernel from persistent data access paths, thereby improving the flexibility and performance. Our simpler programming model improved performance despite the (small) pointer translation overhead. Even a memory hierarchy with large RAM but without persistent memory benefits from our design by enabling programs to operate on large, shared, in-memory data with ease. Our programming model is easy to work with compared to existing systems, and we were able to both quickly prototype real applications with advanced access control features and port existing software (SQLite). Twizzler will give us a system from which we can build a full NVM-based OS around a data-centric design and explore the future of applications, OSes, and processor design on a new memory hierarchy.

Availability Twizzler is available at `twizzler.io`.

Acknowledgements

This work was supported in part by the National Science Foundation (grants IIP-1266400, IIP-1841545), a grant from Intel Corporation, and the industrial members of the UCSC Center for Research in Storage Systems. We thank our shepherd, Yu Hua, the anonymous reviewers, and the members of the Storage Systems Research Center for their support and feedback.

References

[1] The musl C library. https://musl.libc.org/.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 93–112, Atlanta, GA, 1986. USENIX.

[3] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, May 2006. IEEE.

[4] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS '11)*, May 2011.

[5] Piotr Balcer. An introduction to pmemobj (part 1) - accessing the persistent memory. https://pmem.io/2015/06/13/accessing-pmem.html, 2015.

[6] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles (SOSP '69)*, 1969.

[7] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[9] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. A tale of two abstractions: The case for object space. In *Proceedings of HotStorage '19*, July 2019.

[10] Daniel Bittman, Peter Alvaro, and Ethan L. Miller. A persistent problem: Managing pointers in NVM. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS '19)*, pages 30–37, October 2019.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor Mollov, Rajesh Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 385–395, 2010.

[12] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[13] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, pages 191–203, New York, NY, USA, 2017. ACM.

[14] Howard Chu and Symas. Lightning memory-mapped database (part of the OpenLDAP project). https://symas.com/lmdb/.

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 105–118, March 2011.

[16] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Big Sky, MT, October 2009.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, New York, NY, USA, 2010. ACM.

[18] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the November 30 — December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[19] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[20] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds

distributed operating system. *IEEE Computer*, November 1991.

[21] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computer Systems*, 7(3):289–312, June 1994.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High throughput persistent key-value store. In *Proceedings of the 36th Conference on Very Large Databases (VLDB '10)*, September 2010.

[23] Xiangyu Dong, Cong Xu, Norm Jouppi, and Yuan Xie. *Emerging Memory Technologies: Design, Architecture, and Applications*, chapter 2, pages 15–50. Springer, 2014.

[24] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, April 2014.

[25] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 353–368, New York, NY, USA, 2016. ACM.

[26] Dawson R Engler, Sandeep K Gupta, and M Frans Kaashoek. AVM: Application-level virtual memory. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*, pages 72–77. IEEE, 1995.

[27] Dawson R Engler and M Frans Kaashoek. Exterminate all operating system abstractions. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*, pages 78–83. IEEE, 1995.

[28] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, December 1995.

[29] Hewlett Packard Enterprise. YCSB-C. https://github.com/HewlettPackard/meadowlark/tree/master/extra/YCSB-C https://github.com/basicthinker/YCSB-C, 2018.

[30] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[31] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, October 1991.

[32] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 265–278, February 1999.

[33] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[34] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: a distributed single address-space operating system. Technical Report 9314, School of Computer Science and Engineering, University of New South Wales, November 1993.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, pages 295–308, Berkeley, CA, USA, 2011. USENIX.

[36] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 288–303, New York, NY, USA, 1993. ACM.

[37] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 703–717, Santa Clara, CA, June 2017.

[38] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv*, abs/1903.05714, 2019.

[39] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[40] Eric Jul and Bjarne Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, pages 130–132. IEEE, 1991.

[41] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, New York, NY, USA, 1997. ACM.

[42] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 133–145, New York, NY, USA, 2006. ACM.

[43] Dokeun Lee and Youjip Won. Bootless boot: Reducing device boot latency with byte addressable NVRAM. In *2013 International Conference on High Performance Computing*, November 2013.

[44] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 49–64, GA, 2016. USENIX Association.

[45] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *ACM Transactions on Storage*, 12(1), January 2016.

[46] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*, pages 216–223. IEEE, 2014.

[47] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*, Santa Clara, CA, 2017. USENIX Association.

[48] Pankaj Mehra and Samuel Fineberg. Fast and flexible persistence: The magic potion for fault-tolerance, scalability and performance in online data stores. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, January 2004.

[49] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *5th Workshop on Energy-Efficient Design (WEED '13)*, June 2013.

[50] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 401–500, March 2012.

[51] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *Proceedings of the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, July 2018.

[52] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, October 2019.

[53] Matheus Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware-driven undo+redo logging for persistent memory systems. In *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (HPCA 2018)*, February 2018.

[54] Yoann Padioleau and Olivier Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[55] Aleatha Parker-Wood, Darrell D. E. Long, Ethan L. Miller, Philippe Rigaux, and Andy Isaacson. A file by any other name: Managing file names with metadata. In *Proceedings of the 7th Annual International Systems and Storage Conference (SYSTOR '14)*, June 2014.

[56] Timothy Roscoe. Linkage in the Nemesis single address space operating system. *ACM SIGOPS Operating Systems Review*, 28(4):48–55, October 1994.

[57] Andy Rudoff. Persistent memory programming. In *;Login: The Usenix Magazine*, volume 42, pages 34–40. USENIX Association, 2015.

[58] Andy Rudoff et al. Persistent memory programming library. http://pmem.io/nvml/, 2017.

[59] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 59–72, Monterey, CA, June 2002. USENIX.

[61] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 170–185, New York, NY, USA, 1999. ACM.

[62] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical Report 956, University of Wisconsin, August 1990.

[63] Alan Skousen and Donald Miller. Using a single address space operating system for distributed computing and high performance. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC '99)*, pages 8–14, February 1999.

[64] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogenous computing. In *2016 ACM/IEEE 43rd Annual Intenational Symposium on Computer Architecture*, 2016.

[65] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, March 2011.

[66] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, pages 800–812, New York, NY, USA, 2017. ACM.

[67] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[68] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[69] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST '16)*, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

BASTION: A Security Enforcement Network Stack for Container Networks

Jaehyun Nam[†], *Seungsoo Lee*[†], *Hyunmin Seo*[†], *Phillip Porras*[‡],
Vinod Yegneswaran[‡], *and Seungwon Shin*[†]
KAIST, Daejeon, Korea[†], *SRI International, CA, USA*[‡]

Abstract

In this work, we conduct a security analysis of container networks, identifying a number of concerns that arise from the exposure of unnecessary network operations by containerized applications and discuss their implications. We then present a new high-performance security enforcement network stack, called BASTION, which extends the container hosting platform with an intelligent container-aware communication sandbox. BASTION introduces (*i*) a *network visibility service* that provides fine-grained control over the visible network topology per container application, and (*ii*) a *traffic visibility service*, which securely isolates and forwards inter-container traffic in a point-to-point manner, preventing the exposure of this traffic to other peer containers. Our evaluation demonstrates how BASTION can effectively mitigate several adversarial attacks in container networks while improving the overall performance up to 25.4% within single-host containers, and 17.7% for cross-host container communications.

1 Introduction

Among the leading trends in virtualization is that of containerized application deployment at industrial scales across private and public cloud infrastructures. For example, Google has been a significant adopter of container-based software deployment using its container orchestrater, *Kubernetes* [26] to spawn more than two billion containers per week [17]. Yelp uses containers to migrate their code onto AWS, and launches more than one million containers per day [56]. Netflix spawns more than 3 million containers per week within Amazon EC2 using its Titus container management platform [25].

With this growing attention toward the large-scale instantiation of containerized applications also comes a potential for even small security cracks within the container software ecosystem to produce hugely destructive impacts. For example, Tripwire's container security report [50] found that 60% of organizations already had experiences of security incidents in 2018, assessing that these incidents arose primarily due to the pressures to achieve deployment speed over the risks from deploying insecure containers. In recognition of such risks, several efforts [10, 14, 36] have arisen to help identify and warn of possible vulnerabilities in containers.

In addition, the shared kernel-resource model used by containers also introduces critical security concerns regarding the ability of the host OS to maintain isolation once a single container is infected. Indeed, many researchers (and industry) have proposed strategies to address the issue of container isolation. For example, AppArmor [1], Seccomp [40], and SELinux [41] can provide much stronger isolation of containers by preventing various system resource abuses. In fact, several commercial products introduce container security frameworks [2, 44, 51], which can monitor containers at runtime and impose dynamic policy controls.

However, while there continues to emerge a variety of approaches to secure containerized applications, less attention has been paid to bounding these applications' access to the container network. Specifically, there has been significant adoption of containers as microservices [31], in which containers are used to create complex cloud and data-center services. Although current container platforms often utilize IP-based access control to restrict each container's network interactions, there are limitations in such controls that offer opportunities for significant container abuse.

This paper begins by discussing several of the challenges that arise from the current reliance on the host OS network stack and virtual networking features to provide robust container-network security policies. The paper will present five examples of inherent limitations that arise in using the Host OS network stack to manage the communications of container ecosystems as they are deployed today. Informed by these existing limitations, we introduce BASTION, a new extension to container network stack isolation and protection. BASTION instantiates a security network stack per container, offering isolation, performance efficiency, and a fine-grained network security policy specification that implements the least privileged network access for each container. This approach also provides better network policy scalability in network policy management as the number for hosted containers increases, and greater dynamic control of the container ecosystem as containers are dynamically instantiated and removed.

BASTION is composed of a manager and per-container network stacks. The manager solicits network and policy information from active containers, and deploys a security enforcement network stack into each container. Then, in the network stack, all security enforcement is conducted through two major security services: *a network visibility service* and *a traffic visibility service*. Based on a set of inter-container dependencies, the network visibility service mediates the container discovery process, filtering out any access to containers and hosts that are deemed irrelevant given the dependency map. The traffic visibility service controls network traffic between the container and other peer containers, while also verifying the sources of the traffic. This service enables traffic to flow among the containers through an efficient forwarding mechanism that also isolates network communications between senders and recipients. Whenever there is any change in container environments, the manager dynamically updates the network stack of each container with no service interruption.

The paper explains how BASTION mitigates a range of existing security challenges, while also demonstrating that BASTION can improve the overall performance up to 25.4% within the same host and 17.7% across hosts.

Contributions. Our paper contributions are as follows:

• A security assessment of container networks, illustrating security challenges that arise in current container network stacks and security mechanisms.

• The introduction of a novel security-enforcement network stack for containers, which restricts the network visibility of containers and isolates network traffic among peer containers with high performance.

• The presentation of the prototype system, BASTION, including an analysis of how it addresses network security challenges in current container environments.

2 Background and Motivation

Here, we provide the background of container networks and identify how the underlying architectural limitations of current network security services impact container environments.

2.1 Current Container Networks

Docker Platform: Docker [12] uses bridge networks to provide inter-container connectivity, by default. As an example, Figure 1 illustrates the architecture of two microservices. The microservice chains that compose a network service are shown in the upper panel, while the logical networking of the microservice containers, which are networked under separate bridges, is depicted in the lower panel. To provide network flow control, Docker applies network and security policies into bridge networks using `iptables` [34].

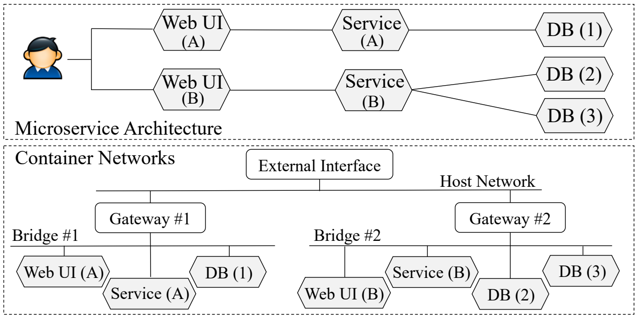


Figure 1: Overview of Docker Bridge Networking. Upper panel: a conceptual microservice architecture involving two independent services. Lower panel: separate bridged networks are instantiated to manage container network flows.

Kubernetes Orchestration System: Kubernetes [26] supports the management of large numbers of Docker containers across multiple nodes (e.g., physical *host* servers), enabling cross-host container applications to work as a logical unit. Thus, while Docker uses bridge networks for containers within the same host (node), Kubernetes uses various overlay networks (e.g., Flannel [11], Weave [55], Calico [49]) to provide inter-container connectivity across multiple nodes. For example, in the Weave overlay network [55], each node has a special bridge interface, called `weave`, to connect all local containers. The weave bridges, run at each node, are logically linked as a single network. While Kubernetes uses Docker containers, it does not utilize Docker networking features to manage network flow control. Rather, it separately applies network policies using `iptables`. Calico [49] similarly applies network and security policies using `iptables`. In the case that operators want further security enforcement, they may use Cilium [7], a security extension that conducts API-aware access control (e.g., HTTP method) by redirecting all network traffic to its containerized security service (envoy).

Network-privileged Containers: Besides the typical use of containers, there are special cases in which an operator wants to directly expose containerized services using the host IP address (e.g., HAProxy [8], OpenVPN [28], and MemSQL [30]). In such cases, by sharing the host namespace with a container, the container is provided access to the host network interfaces, and directly exposes its services. In this work, we refer to such cases as *network-privileged containers*.

2.2 Challenges in Container Networks

While current container platforms mostly utilize OS-level IP-based access control (e.g., `iptables`) to enforce container network security policies, there are significant limitations in their ability to constrain the communication privileges of today's container topologies. The following are five concerns that arise from these current OS-level architectural limitations, which motivate the BASTION design.

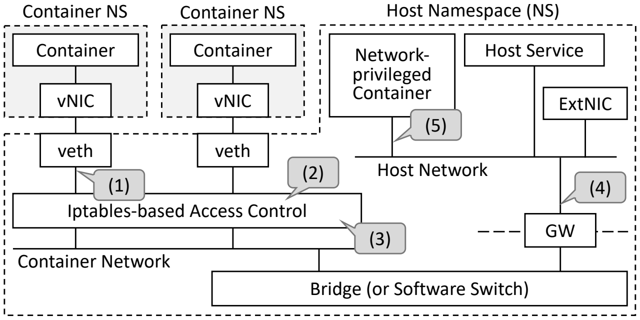
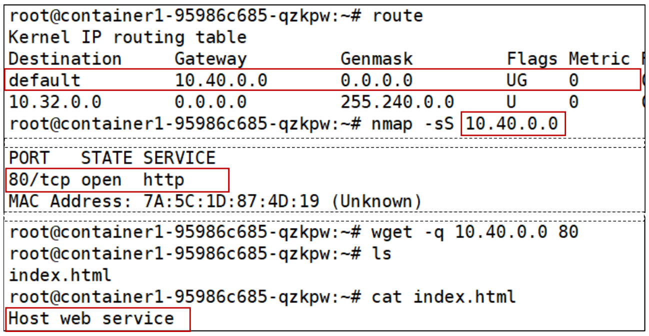


Figure 2: Five critical challenges in container networks: (1) Loss of container context, (2) Limitations of IP-based access controls, (3) Network policy explosion (performance degradation), (4) Unrestricted host access, and (5) No restriction on network-privileged containers.

(1) Loss of container context: As shown in Figure 2, each container has its own virtual interface, but this is only visible inside of the container. Thus, container platforms effectively create a twin virtual interface corresponding to it on a host. This virtual interface is connected to the bridge, enabling connectivity with others. Unfortunately, one security-relevant problem of this design is that each packet produced by a container will lose its association with the source container at the moment that it transitions into the host network namespace, which means that the packet already flows into a container network. Hence, all decisions for further security inspection (e.g., source verification and network flow control) and packet forwarding should be solely made based on the packet header information, and a malicious container can directly forge packets on behalf of any other containers, allowing lateral attacks and traffic poisoning when any container is compromised.

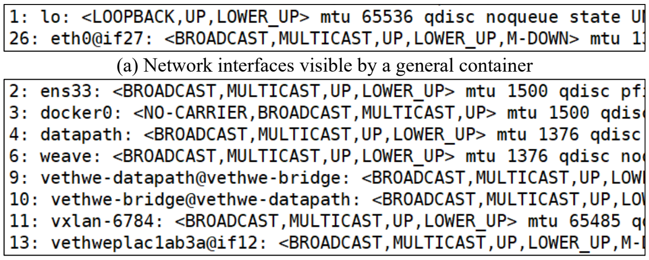
(2) Limitations of IP-based access controls: The primary method for imposing network flow control among container platforms is through `iptables`, an IP-based access control provided by a Linux kernel. However, the IP addresses of containers can be dynamic, and adjustments are then required whenever containers are spun up and down. Thus, it can be a challenge to specify security policies for containers in terms of both performance and security, since these policies must be updated whenever containers are re-created, and the policy tables of `iptables` should be also locked during policy updates. Furthermore, although operators enforce various security policies, container networks are still vulnerable to layer-2 attacks, due to the limited scope of `iptables`.

(3) Network policy explosion: Finer grained network policies inherently require larger sets of network policies. Further, since each container may require different policies, the overall number of policies will tend to increase with the heterogeneity and size of the container ecosystem. Unfortunately, `iptables` is a centralized mechanism for all network interfaces in the host, which results in monolithic network rules that can be daunting to manage and at worst produce



```
root@container1-95986c685-qzkpw:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric
default         10.40.0.0       0.0.0.0         UG    0
10.32.0.0       0.0.0.0         255.240.0.0     U     0
root@container1-95986c685-qzkpw:~# nmap -sS 10.40.0.0
PORT   STATE SERVICE
80/tcp open  http
MAC Address: 7A:5C:1D:87:4D:19 (Unknown)
root@container1-95986c685-qzkpw:~# wget -q 10.40.0.0 80
root@container1-95986c685-qzkpw:~# ls
index.html
root@container1-95986c685-qzkpw:~# cat index.html
Host web service
```

Figure 3: Host service access through the gateway IP address of a container network. A container scans and accesses the services running in the host without any restriction.



```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state U
26: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1
```

(a) Network interfaces visible by a general container

```
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pf
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdis
4: datapath: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc
6: weave: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1376 qdisc no
9: vethwe-datapath@vethwe-bridge: <BROADCAST,MULTICAST,UP,LOW
10: vethwe-bridge@vethwe-datapath: <BROADCAST,MULTICAST,UP,LO
11: vxlan-6784: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65485 q
13: vethwepl ac1ab3a@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-
```

(b) Network interfaces visible by a network-privileged container

Figure 4: Network visibility according to container privileges: upper panel - a general container sees only its own network interfaces, lower panel - a network-privileged container shares the network namespace with a host; thus, it can see all network interfaces in the host.

a network policy explosion (the number of security policies will rapidly increase as a large number of containers are deployed). Consequentially, if the number of security policies in `iptables` increases beyond hundreds, the container ecosystem may face a significant performance degradation [37].

(4) Unrestricted host access: Each container network has a gateway interface for external accesses, which is connected to the host network, as shown in Figure 2. Unfortunately, an inherent security concern arises as a container can thus access a service launched at the host-side. In Kubernetes, containers can even access all other hosts (nodes) through the gateway IP addresses assigned to them. If a service running in a host opens a certain network port, as shown in Figure 3, a container can directly access the service through the gateway IP address. In the worst case, a malicious container can exploit the service in a manner that can subvert/harm the availability of the host.

(5) No restriction on network-privileged containers: While a network-privileged container can gain a performance advantage as its traffic does not pass through additional network stacks (e.g., container networks), such a container also raises significant concerns with respect to operational isolation. As shown in Figure 4, network-privileged containers can access not only the host network interfaces, but can also monitor all network traffic from deployed containers in the

Network Threats	Docker [12]	Flannel [11]	WeaveNet [55]	Calico [49]	Open vSwitch [27]	Cilium [7]	BASTION
L2 attack (e.g., ARP Spoofing)	✔	✔	✔	✘	▲	✘	✘
Traffic Eavesdropping	✔	✔	✔	✘	▲	✘	✘
L3/L4 attack (e.g., IP Spoofing)	✔	✔	✔	✔	▲	▲	✘
Host Service Access	✔	✔	✔	✘	▲	▲	✘
Host Network Namespace Abuse	✔	✔	✔	✔	✔	✔	✘

Table 1: Potential of network attacks across container network interface plugins. Feasible (✔): network attack can be successfully executed over the container network interface plugin. Probable (▲): network attack remains possible, but may be blocked with appropriate application of network security policies. Infeasible (✘): network attack is always blocked.

host and are unrestrained in their ability to inject malicious packets into container networks. Furthermore, current security solutions do not consider security policies for such containers; hence, operators must design and specify a secure policy configuration for the containers by themselves.

2.3 Assumptions and Threat Model

Assumptions: Consider the case of containers connected to each other in order to operate as microservices using Docker or Kubernetes network configurations. Let us assume that an attacker possesses enough skill (e.g., gaining a remote shell to execute arbitrary commands inside a container) to perform a remote hijacking of an Internet-accessible container application that is operating as a part of a microservice, using published container vulnerabilities [45,52]. For example, even certain images provided by the official Docker hub include known vulnerabilities [47]. Given this, we consider what an attacker may do after getting into the subverted container.

Threat Model: The scope of threat models considered in this work focuses on network-based lateral attacks launched from a compromised container, rather than system-based attacks that may occur within a container. Unlike network-based attacks, system-based attacks have been actively explored in other work, such as abusing privileged and unprivileged containers [33] and modifying Linux capabilities within a container [53], and defense techniques based on status inspection of namespaces [24]. Thus, we believe that an operator would properly deploy containers with system-wide security policies, and we therefore do not consider system-wide threats (e.g., attacks against the host kernel) in this paper.

Here, a specific attack case involves one in which a compromised container is employed "as is", as the launching point for these lateral attacks, where *no privilege escalation* is required within the container to conduct further exploitation. Also, an attacker can acquire a base understanding of the compromised container's network configuration by investigating several system files (e.g., `/proc/net/arp`, `/proc/net/route`).

2.4 Limitations of Container Network Interface Plugins

Here, we briefly discuss the limitations of current container networking plugins. Table 1 presents the feasibility of network threats that abuse the above security challenges.

Docker, Flannel, WeaveNet: Docker [12], Flannel [11], WeaveNet [55] operate on bridge-based L2 forwarding, which is tightly coupled with the networking features and the IP-based access control provided by the host OS. Hence, they have the same security challenges discussed in Section 2.2 and are vulnerable to all network threats presented in Table 1.

Calico: Calico [49] employs IP-in-IP-based L3 routing, and uses a single MAC address (`EE:EE:EE:EE:EE:EE`) for all containers which makes L2 attacks infeasible. However, it remains vulnerable to L3/4 attacks (e.g., TCP SYN floods, DNS reflection attacks, ICMP spoofing attacks etc.). In addition, while the host-service abuse is infeasible because Calico uses a virtual gateway IP address (`169.254.1.1`) for all containers, it does not provide security mechanisms that guard against the host-network namespace abuse.

Open vSwitch: Open vSwitch (OVS) [27] provides more flexible networking features than the host OS; thus, it might be viewed as an alternate solution for bolstering container network security. OVS can derive which virtual port a container is connected to and this could be used to prevent spoofing attacks. However, one critical concern is that OVS does not support a `NOT operation`, meaning that we need to install all possible flow rules from each container to other containers, which at least contain (the virtual port and the MAC/IP addresses of a source container, the IP address and the service port of a destination one) matching fields for source verification and spoofing attack prevention. In addition, frequent rule updates are inevitable (as in the case of iptables) whenever containers are spun up and down. While OVS may be able to block unauthorized host IP address accesses, it still allows containers to access host services using gateway IP addresses since OVS is located at the host network namespace. Unfortunately, OVS would still need a large number of security policies against all possible host accesses from each container. In addition, OVS provides no protection in the case of network-privileged containers.

Cilium: Cilium [7] operates at the L3 routing level and provides advanced network security mechanisms for implementing L3-7 firewalls. In addition, L2 attacks are not feasible, as in the case of Calico. However, other network threats remain possible. Although Cilium provides support for a range of network policies (e.g., identity and label-based policies), which can block accesses to specific containers or hosts, the feasibility of such network threats depend on the operator and deployment considerations. For example, even though an operator carefully defines network policies to restrict service-to-service communications based on container identities, containers may still conduct L3/4-based lateral attacks to neighbors in the same service. Network-privileged containers are beyond its threat model, meaning that Cilium is still vulnerable to them.

BASTION: BASTION is designed as a transparent container-network security extension that protects against diverse security challenges discussed in Section2.2. Unlike existing container network interface plugins that rely on operator-defined network policies to protect containers from various network threats, BASTION automatically discovers inter-container dependencies from container platforms, and provides an intelligent container-aware communication sandbox that protects inter-container communications. In the following section, we will describe BASTION in greater detail.

3 BASTION Design

As we discussed in Section 2.2, many of the security limitation that arise from the use of the OS network stack to service container process are less well studied than other container security mechanisms. To address these limitations, we begin by identifying the design considerations that BASTION addresses, followed by a presentation of its design.

R1: *Container-aware least privilege communications enforcement.* A container's connectivity should be a function of the interdependencies between itself and those containers whose communications are required to compose a service.

R2: *Scalable and fine-grained network policy expression.* Network policy expression and enforcement performance within the container network should scale well to the dynamism and size of modern host container topologies.

R3: *Policy control over intra-container communications.* While the gateway interface plays as a key role in the communications with external networks, the network stack should filter out the direct access of the gateway interface to prevent the abuse of the host namespace.

R4: *Policy enforcement for network-privileged containers.* Network policy enforcement should be capable of fine-grained access control over network-privilege-enabled containers that share the host network namespace for the direct access of the host interfaces.

R5: *Unauthorized eavesdropping and spoofing prevention.* Communication mediation should prevent access to third-party packets (i.e., eavesdropping) and false packet production (i.e., preventing both ARP spoofing and traffic injection among local containers).

R6: *Competitive performance that scales well with any container topology.* The network stack should deliver low latency and high throughput communications while securing container networks.

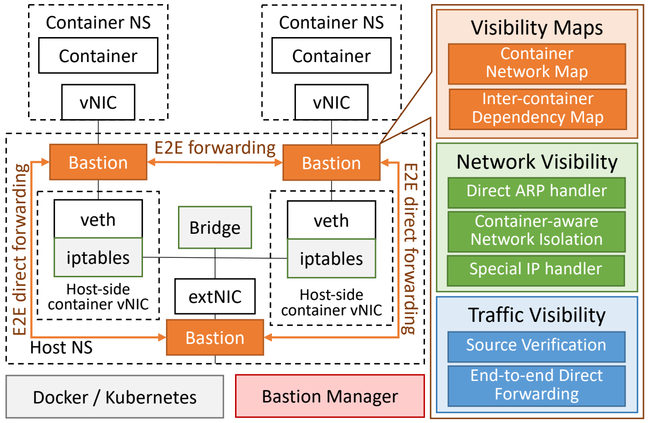


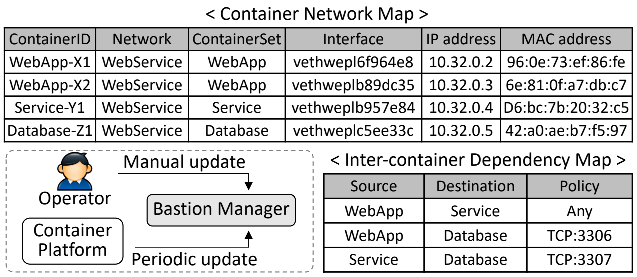
Figure 5: BASTION Architecture Overview. Orange box: BASTION network stack. Red box: manager that maintains the global view of container networks. Green box: network visibility service that restricts container reachability. Blue box: traffic visibility service that controls inter-container traffic while concealing irrelevant traffic from containers.

3.1 Architectural Overview

BASTION represents the opposite spectrum of prior container network stack designs, which implement network policy enforcement in a centralized manner. BASTION implements a decentralized, per-container, network stack. That is, all BASTION security enforcement occurs before a container's packets are delivered into the container network. This approach enables BASTION to provide individualized control over the network traffic coming from each container, mitigating the security challenges discussed in Section 2.2.

Figure 5 illustrates the overall architecture of BASTION. BASTION is composed of a manager, which maintains the global network view of all containers with their security dependencies, and per-container network stacks that include two security services (i.e., network and traffic visibility services). A BASTION network stack maintains the container network map for the corresponding container, which includes the network information of all reachable containers that have peer dependencies (e.g., microservice composition), and an inter-container dependency map, including the security policies on dependent containers only (R2).

When packets arrive at the BASTION network stack, the network visibility service proactively filters any discovery processes of irrelevant containers by dealing with ARP requests based on the container network map (R1, R5), and restricts the communications between containers according to security policies specified in the inter-container dependency map (R1). In addition, a special IP-handler restricts unauthorized access to special IP addresses (e.g., gateway IP addresses) (R3). The traffic visibility service conducts secure packet-forwarding between containers. This service first verifies the packets with the identity of the container (R4-5), directly passing packets



< Container Network Map >

ContainerID	Network	ContainerSet	Interface	IP address	MAC address
WebApp-X1	WebService	WebApp	vethwepl6f964e8	10.32.0.2	96:0e:73:ef:86:fe
WebApp-X2	WebService	WebApp	vethweplb89dc35	10.32.0.3	6e:81:0f:a7:db:c7
Service-Y1	WebService	Service	vethweplb957e84	10.32.0.4	D6:bc:7b:20:32:c5
Database-Z1	WebService	Database	vethweplc5ee33c	10.32.0.5	42:a0:ae:b7:f5:97

< Inter-container Dependency Map >

Source	Destination	Policy
WebApp	Service	Any
WebApp	Database	TCP:3306
Service	Database	TCP:3307

Figure 6: BASTION computes a container network map that captures the network interface attributes for each hosted container, and an inter-container dependency map that indicates the links and dependencies between containers.

Algorithm 1 Extracting Inter-Container Dependencies

```
Input: $C$, which is the set of all active containers
for each container $u \in C$ do
    for each container $v \in C$ where $u \neq v$ do
        if $v \in u.explicitDependents$ then
            $p_{uv} = u.EgressPolicies \cap v.IngressPolicies$
            add $v$ into $u.dependencyMap$ with $p_{uv}$
        else if $u.containerSet \neq v.containerSet$ then
            for each service pair $s \in S_{set}(v.ContainerSet)$ do
                $p_{us} = u.EgressPolicies \cap s.Port$
                add $s.IP$ into $u.dependencyMap$ with $p_{us}$
    for each service pair $s \in S_{microservice}$ do
        $p_{us} = u.EgressPolicies \cap s.Port$
        add $s.IP$ into $u.dependencyMap$ with $p_{us}$
```

from the source container to the destination containers using their interface information (R6). Since this direct packet forwarding occurs at the network interface level, packets are no longer passed through the container network (host-side), eliminating any chance for unauthorized network traffic exposure (even to network-privileged ones) (R4-5).

In terms of cross-host inter-container communications, a specialized BASTION network stack is utilized at the external interface of each node. It only maintains the container network map for all containers deployed in each node since all security decisions are already made at the network stack of each container. Thus, when it receives packets from other nodes, it simply conducts a secure forwarding from the external interface to destination containers. Overlay network composition among hosts (nodes) are beyond the coverage of BASTION; thus, it utilizes existing overlay networks (e.g., WeaveNet over IPSec). BASTION also retains the existing mechanisms of container platforms to handle inbound traffic from external networks.

3.2 BASTION Manager

The BASTION manager performs two primary roles. It collects the network information of all active containers from container platforms and manages the BASTION network stacks deployed to the active containers.

(1) Container Collection. The BASTION manager first maintains two hash maps (i.e., a global container network map and the inter-container dependency map for all containers) for the security evaluation of each container. As shown in Figure 6, BASTION uses a container platform to retrieve the network information for all containers, and to build the inter-container dependency map by extracting the dependencies among containers based on the retrieved information and their ingress/egress security policies. In addition, because containers can be dynamically spun up and down, the manager periodically retrieves containers' network information to update the maps. While a notification-based mechanism would provide greater efficiency, a polling-based mechanism was selected to provide a transparent and compatible solution that can be integrated with already-deployed container environments without any required modifications.

Extracting inter-container dependencies: BASTION automatically extracts dependencies among containers. To do this, a container network model is defined, in which a microservice is composed of one or more container sets, and each container set has one or more containers that play the same role (due to scaling and load-balancing). Each container set exposes internal service ports to communicate with other container sets, while a microservice exposes global service ports to redirect accesses from the outside world to some of the internal service ports. We then define four constraints for implicit dependencies in inter-container communications: (1) containers with the same container set are not granted interconnectivity, (2) containers in different container sets only communicate via internal service ports (explicitly exposed by configurations), (3) containers that are unrelated to each other may talk through global service ports, (4) all other communications are not allowed by default. Based on the container network model, all inter-container dependencies are extracted using Algorithm 1.

Discovering inter-container dependencies: As no container network can be made secure without proper network policies that restrict communications to the minimum required access, BASTION also discovers inter-container dependencies not explicitly defined by a container operator. During the flow control of inter-container traffic, BASTION produces network logs that capture the network accesses from/to containers. At the same time, it compares these logs with the inter-container dependency map, classifying them into three cases: *legitimate accesses, missing policies, and excessive policies.*

If the pair of observed containers are not in the precomputed inter-container dependency map, BASTION considers that there is either a missing network policy or an invalid access. Then, it informs an operator to review the specific flows to determine whether to produce a missing network policy. In addition, it identifies network policies for which no flows have been encountered. Such cases may represent an over-specification of policies that enable unnecessary flows for the container network's operations. In these cases, BASTION informs an operator to review the specific policy that may require to be updated in the current configuration.

(2) Network Stack Management. The manager maintains the BASTION secure network stack for each container. For newly spawned containers, it installs the network stacks at their interfaces with the container network and inter-container dependency maps. With respect to map size, each container only requires a part of the network information to communicate with dependent neighbors. Thus, to reduce the size of security services, BASTION filters irrelevant information per container. The manager also performs change detection of inter-container dependencies, automatically updating the maps in the network stacks of the corresponding containers.

3.3 Network Visibility Service

The network visibility service restricts unnecessary connectivity among containers and between containers and external hosts. To do this, the following three security components are introduced to handle container discovery, inter-container communications, gateway/service-IP accesses, respectively.

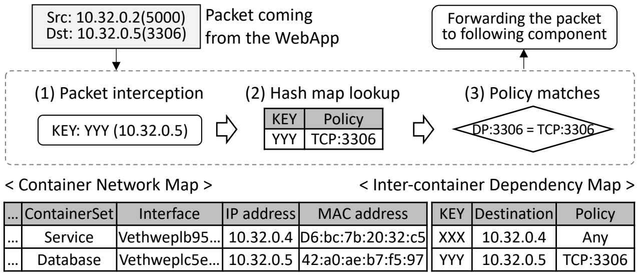
3.3.1 Direct ARP Handler

For inter-container networking, container discovery is the first step to identify other containers (communication targets). Containers use ARP requests to identify the necessary network information (i.e., MAC addresses) of target containers. Unfortunately, this discovery process can be exploited to scan all containers connected to the same network by malicious containers, as current network stacks do not prevent ARP scan. Indeed, they offer no mechanism to control non-IP-based communications.

BASTION's direct ARP handler filters out any unnecessary container discovery that does not pertain to the present container's dependency map. When a container sends an ARP request, the handler intercepts the request before it is broadcasted, verifying if the source container has a dependency on the destination container. This analysis is done using the inter-container dependency map. If accessible, the handler generates an ARP reply with the MAC address of the destination container in the container network map, and sends the reply back to the source container (while no ARP requests flow into the container network). If not, it drops the request.

3.3.2 Inter-container Communications Handler

Although the direct ARP handler prevents containers from performing unbounded topology discovery, its coverage is limited to container-level isolation. It does not address malicious accesses among dependent containers. Hence, to further restrict the reachability of containers, a second component implements container-aware network isolation.



KEY	Policy
YYY	TCP:3306

...	ContainerSet	Interface	IP address	MAC address
...	Service	Vethweplb95...	10.32.0.4	D6:bc:7b:20:32:c5
...	Database	Vethweplc5e...	10.32.0.5	42:a0:ae:b7:f5:97

KEY	Destination	Policy
XXX	10.32.0.4	Any
YYY	10.32.0.5	TCP:3306

Figure 7: Workflow of container-aware network isolation. The WebApp container accesses a service of the Database container in a container network shown in Figure 6, and the container-aware network isolation in the WebApp's network stack inspects their dependency and security policies.

To illustrate how BASTION implements container-aware network isolation, we consider the example in Figure 6, which illustrates an interdependence between WebApp and Database containers. In mediating the WebApp's packets, as shown in Figure 7, BASTION first checks the dependency between the WebApp and the destination by examining the inter-container dependency map using the destination IP address as a key. If any policies exist in the map, it concludes that the WebApp has a dependency on the destination - in this case the Database. The connection is allowed if matched to the policy for the Database, otherwise it is dropped.

BASTION implements a per-container rule partitioning strategy, which simplifies rule conflict evaluation, as only the container-relevant rules are considered, at deployment and evaluation times. In addition, it offers a minimized policy enforcement performance impact, as the match set is container-specific rather than host-global (as occurs with `iptables`). This approach offers an inherent key advantage over host global network policy rule enforcement as the number of containers increases. A natural strategy for managing large sets of global (host layer) network security rules is to apply a global rule optimization algorithm (e.g., aggregating the rules into a reduced set). Unfortunately, as containers are dynamically spun up and down, particularly within large orchestrated container ecosystems, their corresponding security rules would also require frequent updating. In such situations, global rule optimization could prove less effective and even be a new performance bottleneck over our rule partitioning strategy.

3.3.3 Gateway and Service-IP Handler

In container environments, it is possible for a subverted container to exploit the gateway to probe services within the host OS. To address this concern, BASTION's gateway-IP handler filters direct host accesses. When a network connection targets non-local container addresses, it includes the gateway MAC address and the IP address of the actual destination. Based on this fact, the gateway-IP handler blocks any direct host accesses by checking if both IP and MAC addresses be-

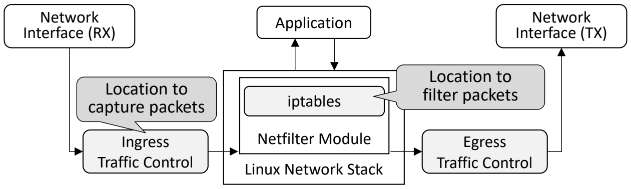


Figure 8: An illustration of the network packet processing sequence performed within the Linux kernel. While packets are filtered after delivered into the network stack, those can be still exposed during packet capture with nothing missed.

long to the gateway. It would be also possible that a network flow might access the gateways of other container networks, since these gateways are connected to the host network as well. Hence, the gateway-IP handler also filters unauthorized host accesses by comparing packets with the other gateways.

In Kubernetes environments, there is another special IP address, called a service IP address that is a virtual IP address used for the redirection to actual containers. Unfortunately, since service IP addresses do not belong to container networks, they can be simply considered as external IP addresses. Thus, BASTION additionally extracts the pairs of {service IP address, port} and {corresponding container IP address, port} from Kubernetes, and maintains a service map in each BASTION network stack. Then, when a container sends a packet with a service IP address and port, the service-IP handler overwrites the service IP address and port to an actual container IP address and port according to the service map. As a result, all inter-container communications can be conducted with the existent IP addresses, and the other security components can process packets as intended.

3.4 Traffic Visibility Service

The traffic visibility service provides point-to-point integrity and confidentiality among container network flows. Here, we present how BASTION hides irrelevant traffic from containers using two security components: source verification and end-to-end direct forwarding.

3.4.1 Source Verification

To precisely track the actual source of inter-container traffic, BASTION leverages the kernel metadata of incoming packets (e.g., ingress network interface index). The BASTION network stack of each container statically contains the network information (i.e., IP/MAC addresses and the metadata of container-side and host-side interfaces) of the corresponding container, and BASTION verifies the incoming traffic by comparing not only the packet header information but also its metadata to the container's information embedded in the BASTION network stack. If either the packet header information or the metadata is not matched with the container network information, BASTION identifies the incoming traffic as spoofed

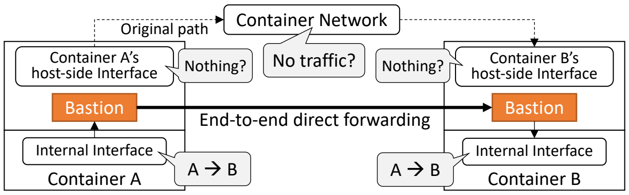


Figure 9: An illustration of how BASTION implements end-to-end direct packet forwarding to bypass exposure of intra-container traffic to other containers.

and drops it. Furthermore, even though network-privileged containers can inject spoofed packets into other containers, BASTION will drop their spoofed packets since the packet metadata would not be matched with the container network information. As a result, BASTION can effectively eliminate the spectrum of disruption and spoofing threats.

3.4.2 End-to-end Direct Forwarding

Current network stacks cannot prevent the exposure of inter-container traffic from other containers as the filter position is behind the capture point, as illustrated in Figure 8. Thus, if a malicious container has a capability to redirect the traffic of a target container to itself, it can monitor the traffic without restriction. In the case of network-privileged containers, they have the full visibility of all container networks: they can directly monitor the network traffic of others with no need to redirect the traffic.

To implement least-privilege traffic exposure, BASTION provides an end-to-end direct forwarding component. As shown in Figure 9, this component performs direct packet delivery between source and destination containers in the network interface level, bypassing not only their original network stacks (container-side) but also bridge interfaces (host-side); thus, it can prevent eavesdropping by peer containers. As soon as BASTION receives an incoming network connection from a container, it retrieves the interface information of a destination from the container network map. If the destination is a container in the same node, BASTION directly injects the packet stream into the destination container. If the destination is a container in another node, BASTION injects the packet to the external interface of a host. Then, once the special BASTION network stack of the external interface at the target node receives the packet, it directly injects the packet stream into the destination container. This traffic isolation prevents any traffic disclosure by other containers, preventing even network-privileged containers to view third-party traffic.

4 Implementation

We implement BASTION with 2.2K lines of C code and 5.1K lines of Python code on the Linux 4.16 kernel, which include two subsystems: a manager, and a network stack.

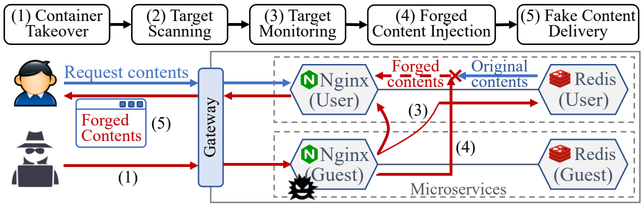


Figure 10: An example attack scenario within a Kubernetes environment. A compromised container from one service conducts a series of network attacks to hijack communications between other containers in a peer service.

BASTION Manager: For container collection, the manager periodically captures the attributes (e.g., NetworkSettings) of active containers from the Docker engine and the Kubernetes API server. Especially, in terms of explicit inter-container dependencies, it utilizes specific keywords (e.g., "*link*" and "*depends_on*"). In the case of Kubernetes, it does not have a way to explicitly define inter-container dependencies; thus, the manager utilizes `labels` to define explicit inter-container dependencies (e.g., "*dependencies: container A*"). In terms of network security policies, it extracts "*ingress*" and "*egress*" network security policies from `iptables` in a host and Kubernetes.

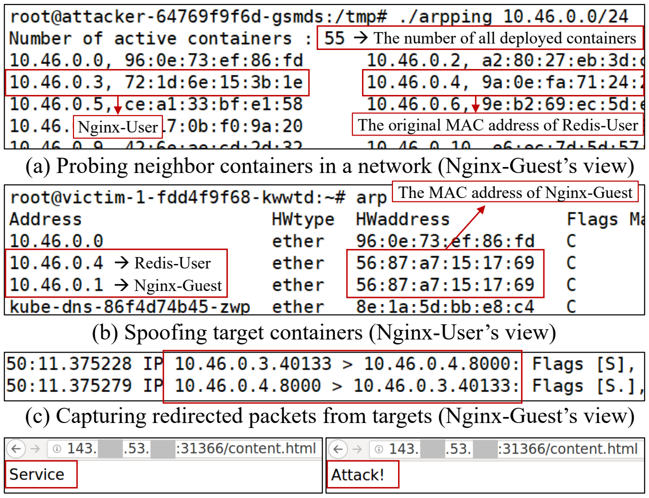
BASTION Network Stack: The security enforcement network stack for each container is implemented using eBPF [22] and XDP [19, 21], and the security services in the network stack inspect raw incoming packets in the `xdp_md` structure provided by XDP. During the inspection, they look up two hash maps (i.e., the container network and inter-container dependency maps), and these maps are synchronized with the maps in the corresponding management thread of the manager using BPF syscalls. Then, they use three types of XDP actions: 'XDP_TX' sends a packet back to the incoming container (the direct ARP handler), 'XDP_REDIRECT' injects a packet into the transmit queue of a destination (the end-to-end direct forwarding), and 'XDP_DROP' drops packets.

5 Security Evaluation

This section introduces a scenario that abuses the security holes in the current container network with real containers, and demonstrates how BASTION mitigates network attacks.

5.1 Scenario Validation

Figure 10 illustrates two independent services that are deployed along with common microservices [20, 54] in a Kubernetes environment. One is a service for legitimate users, and the other is a service for guest users. These services use Nginx [35] and Redis [38] container images retrieved from Docker Hub [13]. In this scenario, an attacker forges legitimate user requests, after infiltrating into the public-facing Nginx server by exploiting web application vulnerabilities.



```
root@attacker-64769f9f6d-gsmds:/tmp# ./arpping 10.46.0.0/24
Number of active containers : 55 → The number of all deployed containers
10.46.0.0, 96:0e:73:ef:86:fd       10.46.0.2, a2:80:27:eb:3d:c
10.46.0.3, 72:1d:6e:15:3b:1e       10.46.0.4, 9a:0e:fa:71:24:2
10.46.0.5, ce:a1:33:bf:e1:58       10.46.0.6, 9e:b2:69:ec:5d:e
10.46. Nginx-User .7:0b:f0:9a:20   The original MAC address of Redis-User
10.46.0.9, 42:6e:ae:cd:2d:32       10.46.0.10, e6:ec:7d:5d:57
```

(a) Probing neighbor containers in a network (Nginx-Guest's view)

```
root@victim-1-fdd4f9f68-kwwtd:~# arp   The MAC address of Nginx-Guest
Address                  HWtype  HWaddress           Flags Ma
10.46.0.0                ether   96:0e:73:ef:86:fd   C
10.46.0.4 → Redis-User   ether   56:87:a7:15:17:69   C
10.46.0.1 → Nginx-Guest  ether   56:87:a7:15:17:69   C
kube-dns-86f4d74b45-zwp  ether   8e:1a:5d:bb:e8:c4   C
```

(b) Spoofing target containers (Nginx-User's view)

```
50:11.375228 IP 10.46.0.3.40133 > 10.46.0.4.8000: Flags [S],
50:11.375279 IP 10.46.0.4.8000 > 10.46.0.3.40133: Flags [S.],
```

(c) Capturing redirected packets from targets (Nginx-Guest's view)

143.[redacted].53.[redacted]:31366/content.html

Service

143.[redacted].53.[redacted]:31366/content.html

Attack!

(d) Injecting packets with forged contents (before / after)

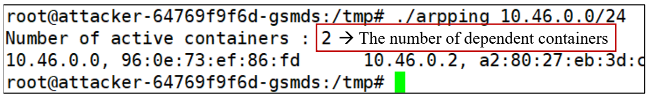
Figure 11: Screenshots demonstrating the attack scenario in a Kubernetes environment between two services.

In this attack kill chain, the attacker leverages three network-based attacks to compromise the Nginx-Guest container and successfully execute a man-in-the-middle attack. In the first step, he discovers active containers around the network through ARP-based scanning. Since all containers are connected to an overlay network and ARP packets are not filtered by `iptables`, the attacker can easily collect the network information of containers as shown in Figure 11-(a). Then, the attacker injects fake ARP responses into the network to make all traffic between the Nginx-User and the Redis-User containers passes through the Nginx-Guest. As shown in Figure 11-(b), we can see that the MAC address of the Redis-User in the ARP table of the Nginx-User is replaced with that of the Nginx-Guest, and the attacker monitors all traffic between the Nginx-User and the Redis-User (Figure 11-(c)). Lastly, the attacker replaces the response for the legitimate user with forged contents by internally dropping the packets delivered from the Redis-User and injecting forged packets. Then, the Nginx-User returns the forged contents back to the user instead of the original ones (Figure 11-(d)). In the end, the user receives forged contents as the attacker intended.

5.2 Effectiveness of Security Functions

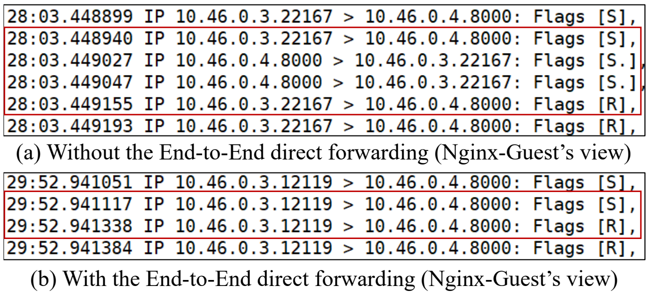
Here, we focus on validating the effectiveness of BASTION against a range of network-oriented attacks. *For the following experiments, we disabled some of BASTION's security functions to show the before and after differences.*

Container Discovery: When a compromised container is used to conduct peer discovery to locate other containers, as shown in Figure 11-(a), the current container network stack allows an attacker to discover all neighboring containers. On the other hand, as shown in Figure 12, BASTION's



```
root@attacker-64769f9f6d-gsmds:/tmp# ./arpping 10.46.0.0/24
Number of active containers : 2 → The number of dependent containers
10.46.0.0, 96:0e:73:ef:86:fd      10.46.0.2, a2:80:27:eb:3d:c
root@attacker-64769f9f6d-gsmds:/tmp#
```

Figure 12: An illustration of neighbor container discovery. Our ARP handler and container-aware flow control only allow the inter-dependent containers to be shown.



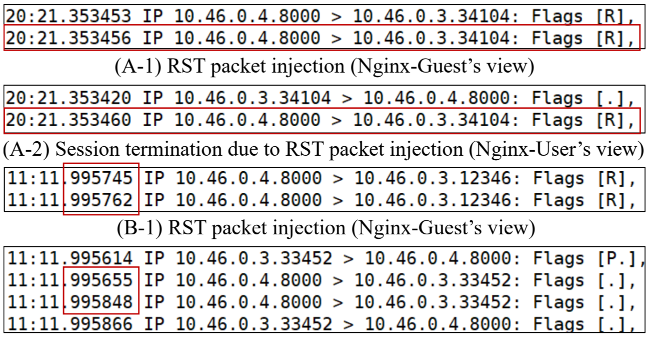
```
28:03.448899 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [S],
28:03.448940 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [S],
28:03.449027 IP 10.46.0.4.8000 > 10.46.0.3.22167: Flags [S.]
28:03.449047 IP 10.46.0.4.8000 > 10.46.0.3.22167: Flags [S.]
28:03.449155 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [R],
28:03.449193 IP 10.46.0.3.22167 > 10.46.0.4.8000: Flags [R],
```

(a) Without the End-to-End direct forwarding (Nginx-Guest's view)

```
29:52.941051 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [S],
29:52.941117 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [S],
29:52.941338 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [R],
29:52.941384 IP 10.46.0.3.12119 > 10.46.0.4.8000: Flags [R],
```

(b) With the End-to-End direct forwarding (Nginx-Guest's view)

Figure 13: Restricting Traffic Visibility: upper panel - an attacker can see the traffic of the spoofed target container without end-to-end forwarding, lower panel - the attacker cannot see response traffic with end-to-end forwarding (only the reverse direction is intentionally filtered for illustration).



```
20:21.353453 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
20:21.353456 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
```

(A-1) RST packet injection (Nginx-Guest's view)

```
20:21.353420 IP 10.46.0.3.34104 > 10.46.0.4.8000: Flags [.],
20:21.353460 IP 10.46.0.4.8000 > 10.46.0.3.34104: Flags [R],
```

(A-2) Session termination due to RST packet injection (Nginx-User's view)

```
11:11.995745 IP 10.46.0.4.8000 > 10.46.0.3.12346: Flags [R],
11:11.995762 IP 10.46.0.4.8000 > 10.46.0.3.12346: Flags [R],
```

(B-1) RST packet injection (Nginx-Guest's view)

```
11:11.995614 IP 10.46.0.3.33452 > 10.46.0.4.8000: Flags [P.],
11:11.995655 IP 10.46.0.4.8000 > 10.46.0.3.33452: Flags [.],
11:11.995848 IP 10.46.0.4.8000 > 10.46.0.3.33452: Flags [.],
11:11.995866 IP 10.46.0.3.33452 > 10.46.0.4.8000: Flags [.],
```

(B-2) Invisible injected RST packets (Nginx-User's view)

Figure 14: Restriction of Packet Injection. Panel A-1 shows the attacker injecting RST packets, and A-2 shows the victim session terminated by attacker's RST packet. Panel B-1 shows the trials of RST packet injections, and B-2 shows the failure of RST packet injection due to source verification.

direct ARP handler and container-aware network isolation reduce the reachability of each container based on its container dependencies (R1). As a result, the infected container (i.e., Nginx-Guest in Figure 10) has only one dependent container (i.e., Redis-Guest in Figure 10), and BASTION ensures that the container observes only its gateway and that dependent.

Passive Packet Monitoring: As discussed previously, a compromised container may be able to sniff the network traffic of a target container. Further, when an attacker compromises a "network-privileged" container, the attacker is provided access to all network traffic, with no restriction. BASTION mitigates these concerns by implementing end-to-end direct container traffic forwarding.

Figure 13 illustrates the utility of BASTION's direct forwarding. The upper panel, Figure 13-(a), shows the visible network traffic of a target container (i.e., Nginx-User) after spoofing the container without direct forwarding. The lower panel, Figure 13-(b), demonstrates the use of direct forwarding. When direct forwarding is applied, the only visible traffic from a given interface is that of traffic involving the container itself (R4, R5). *To highlight the differences, we intentionally make the flow from a source to a destination visible.* As a result, while the attacker can observe the source-to-destination flow, he can no longer observe the traffic in the reverse direction. If we fully apply end-to-end forwarding for all traffic, the attacker will see no traffic between them.

Active Packet Injection: Network-based attacks frequently rely on spoofed packet injection techniques to send malicious packets to target containers. BASTION prevents these attacks by performing explicit source verification. To illustrate its impact, we demonstrate before and after cases from the attacker and victim perspectives. In the following example, we enable source verification only at the IP-level and allow an attacker to conduct ARP spoofing attacks.

Figure 14-A illustrates cases without source verification. Here, the attacker spoofs the Nginx-User and receives the traffic of the Nginx-User. Further, the attacker injects RST packets to terminate the session of the Nginx-User. As soon as the attacker injects the RST packets, as shown in panel A-2, the Nginx-User receives the injected RST packets (see the received times of the RST packets), causing its session to be immediately terminated. This situation is remedied with explicit source verification. Although the attacker tries to inject RST packets, as shown in panel B-2, the RST packets are rejected by the source verification component and prevented from reaching the Nginx-User (R5).

6 Performance Evaluation

This section summarizes our measurement results of BASTION's performance overhead with respect to latencies and throughputs between containers under various conditions.

Test Environment: We used an experimental testbed comprising of three machines to construct a Kubernetes environment with the Weave overlay network and evaluate the BASTION prototype. One system served as the Kubernetes master node, while the others acted as container-hosting nodes. Each system was configured with an Intel Xeon E5-2630v4 CPU, 64 GB of RAM, and an Intel 10 Gbps NIC. `netperf` [18] and `iperf` [23] were respectively used to measure round-trip latencies and TCP stream throughputs.

6.1 Network Stack Deployment Overhead

BASTION periodically retrieves the container information from container platforms (1 second in our case) and deploys the network stacks for newly detected containers.

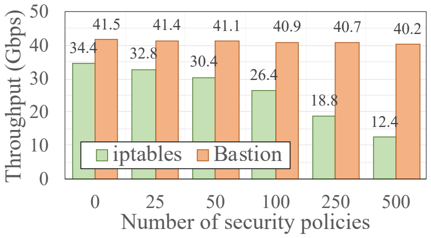


Figure 15: Inter-container throughput variations with the increasing number of security policies within a host.

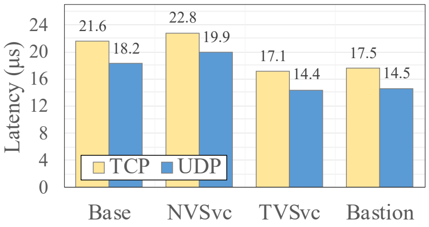


Figure 16: Inter-container latency measurements with different combinations of BASTION's services within a host.

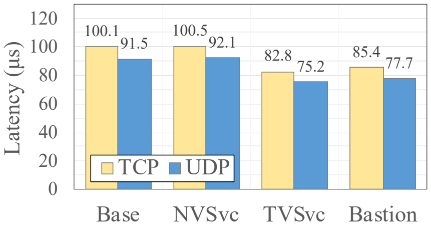


Figure 17: Inter-container latency measurements with different combinations of BASTION's services across hosts.

To see how long it takes to deploy a new network stack, we measured the deployment time while creating 100 containers. The result shows that it took 13.03 μs on average, meaning that BASTION's network stack would be deployed almost right after a new container is detected, while it can take a couple of seconds for containers to initialize their services (i.e., pulling container images from repositories, configuring container isolation, and starting services).

6.2 Security Policy Inspection Overhead

We compared the matching overheads with both `iptables`-based access control and BASTION, and Figure 15 shows the TCP throughputs with different numbers of security policies. For a fair comparison, we defined the same number of policies to each container for the overhead measurements of BASTION.

In the case of `iptables`, security policies for all containers are maintained collectively in the host kernel. Thus, when packets arrive from containers, `iptables` first looks up the policies for the corresponding containers and inspects them individually with the incoming packets. Also, `iptables` requires a large number of field matches (at least, source and destination IP addresses and ports for each policy) since it is designed for general access control. As a result, as shown in Figure 15, the throughput degraded by 23.3% with 100 policies and 64.0% with 500 policies. This trend points to a fundamental scaling challenge with the current policy enforcement approach for container networks. In contrast, the throughput degradation caused by BASTION was barely noticeable as the number of policies increased (3.2% with 500 policies) (R2). Such performance gains stem from BASTION's matching process optimized for containers, which comprises of a hash-based policy lookup for specific destinations and their port matches while there is no need to match source IP addresses and ports. Note that BASTION's performance gain with no security policy is because of the end-to-end direct forwarding that bypasses the host-side Linux network stack.

6.3 Performance: Single-Host Deployment

Here, we evaluated latencies and throughputs between containers hosted in the same node to measure the overhead of BASTION. Figure 16 provides the round-trip latency comparison of four test cases within a single node. The base case provides latency measurements for a default configuration of two containers that interacted with no security services, which were 21.6μs and 18.2μs for TCP and UDP packets respectively. When we applied BASTION's network visibility service, the latencies slightly increased by 5.7% and 9.3% due to the newly applied security functions requiring additional packet processing to derive the reachability check between containers. When we applied BASTION's traffic visibility service, the overall latencies were noticeably improved by 26.3% because our secure forwarding directly fed inter-container traffic into destination containers while bypassing the existing container networks. Finally, we observed the overall performance improvement with respect to the base case of 23.0% and 25.4% for TCP and UDP packets when all BASTION security functions were fully applied (R6). Table 2 also shows that the overall throughput of BASTION was improved by 20.6% compared to that of the base case within a host.

Throughput (Gbps)	Base	Network Visibility	Traffic Visibility	Bastion
Within a host	34.4	33.7	41.8	41.5
Across hosts	4.28	4.23	4.91	4.83

Table 2: Inter-container throughput measurements for the base case (no security) and BASTION's services.

6.4 Performance: Cross-Host Deployment

Next, we measured the latencies and throughput for cross-host container deployments. Figure 17 illustrates the measurement results with different combinations of BASTION's security services. Compared to the intra-host measurements, the overall latencies significantly increased due to physical link traversal and tunneling overheads between hosts; thus, the latency of the base case became 100.1μs and 91.5μs for TCP and UDP packets respectively. Also, given the network impact, the overhead caused by BASTION's network-visibility service receded (less than 1%). Next, when we introduced BASTION's traffic-visibility service, the latencies were reduced by 21.3%

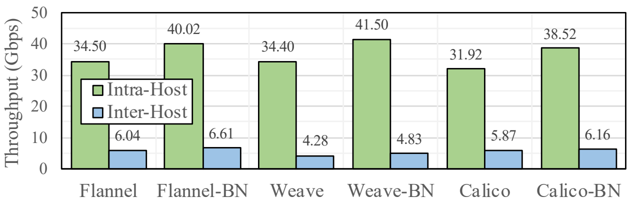


Figure 18: Throughput comparison with different types of container networks. BN = (when BASTION is deployed).

on average because our secure forwarding directly passed network packets from the source container to the destination container via the external interfaces. Finally, when we applied all security services, the latencies decreased by 17.7%, a significant improvement compared to the base case (R6). These latency improvements translated to a cross-host throughput improvement of 12.9%, as shown in Table 2.

6.5 Performance: Networking Plugins

Lastly, we compared the throughput variations in different types of container networks when BASTION is deployed. Figure 18 shows the TCP-stream throughputs between intra-host and inter-host containers in three container networks (i.e., Flannel, WeaveNet, and Calico). From the results, we see that the intra-host throughputs are improved 16.0% in the Flannel network, 20.6% in the Weave network, and 20.7% in the Calico network through BASTION. In terms of the inter-host throughputs, we also see the performance improvements (9.4%, 12.9%, and 4.9% respectively) through BASTION. In sum, we ascertain the fact that BASTION can provide not only further network isolation and security enforcement but also better performance in various container networks.

7 Related Work

Container Security Analysis. There have been several efforts [15, 16, 24, 32, 33, 53] that have analyzed the security issues of container implementations. For example, Dua et al. [15] analyzed various container implementations, concluding that they are yet insecure from the perspective of the filesystem, network, and memory isolation. More specifically, Jian et al. [24] demonstrated a Docker *escape attack*, which allows an adversary to break out of the isolation of a Docker container by exploiting a Linux kernel vulnerability. Another research area [43, 45, 47, 48, 52] of container security focuses on container images. Shu et al. [43] and Tak et al. [47, 48] have performed a large-scale vulnerability assessment of Docker images on Docker Hub and shown that many images were outdated and vulnerable. While these studies broadly point out the security issues of containers, their goals differ from our work. Rather, BASTION focuses on container networks.

Container Security and Isolation. Bacis et al. introduced DockerPolicyModules (DPM) [4] that allow Docker image maintainers to specify and ship SELinux policies within their images. Sun et al. [46] proposed security namespaces that enable containers to independently define security policies and apply them to a limited scope of processes. SCONE [3] presented a secure container mechanism for Docker containers by isolating them inside of SGX enclaves. LightVM [29] wraps containers in lightweight virtual machines (VMs). X-Containers [42] isolate containers that have the same concerns together on top of separate library OSes. These efforts are complementary to the network-focused objectives of BASTION, and could be combined to deliver security services that span both the system and networking services.

Container Network Security. Most container network solutions [57, 58] have focused on container network performance, with little attention to fine-grained policy enforcement. A few recent studies investigated the security issues in container networks. Bui [5], Comb et al. [9] and Chelladhurai et al. [6] analyzed Docker container security, finding that Docker is vulnerable to ARP spoofing and MAC flooding attacks in default settings. Our work extends these results by identifying broader class of attacks, and we present system extensions that address these problems. With respect to security policies for inter-container communications, while most solutions [39, 49, 55] have adopted `iptables`, Cilium [7] provides its own API-aware security mechanisms for L3/4/7 policies. As we discussed previously, while Cilium pursues API-level network security filtering to define and enforce both network- and application-layer security policies, BASTION fundamentally redesigns a network stack per container to construct an inherently secure container networking system.

8 Conclusion

Containerization has emerged as a widely popular virtualization technology that is being aggressively deployed into large-scale enterprise and cloud environments. However, this adoption could be stifled by critical security issues, which remain understudied. We have analyzed the security challenges involved in the current container network stack, and addressed these challenges by presenting BASTION, an intelligent communication sandbox for securing container-network communications, using Linux kernel features. In this work, we raise awareness of several security problems that lie within today's container networks, and offer security services for halting these problems in real-world container deployments.

Acknowledgement

We would like to thank all the anonymous reviewers of the program committee for their valuable insights on the paper. This work was partially funded by the National Science Foundation (NSF) under Grant No. CNS-1514503.

References

[1] AppArmor. https://gitlab.com/apparmor.

[2] Aqua. https://www.aquasec.com.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[4] Enrico Bacis, Simone Mutti, Steven Capelli, and Stefano Paraboschi. DockerPolicyModules: Mandatory Access Control for Docker Containers. In *Proceedings of the Conference on Communications and Network Security*. IEEE, 2015.

[5] Thanh Bui. Analysis of Docker Security. *arXiv preprint arXiv:1501.02967*, 2015.

[6] Jeeva Chelladhurai, Pethuru Raj Chelliah, and Sathish Alampalayam Kumar. Securing Docker Containers from Denial of Service (DoS) Attacks. In *Proceedings of the International Conference on Services Computing*. IEEE, 2016.

[7] Cilium. API-aware Networking and Security. https://cilium.io.

[8] CiscoCloud. HAProxy Docker Container. https://hub.docker.com/r/ciscocloud/haproxy-consul.

[9] Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or Not to Docker: A Security Perspective. *Proceedings of the Cloud Computing*, 2016.

[10] CoreOS. Clair. https://coreos.com/clair/docs/latest.

[11] CoreOS. Flannel. https://coreos.com/flannel.

[12] Docker. https://www.docker.com.

[13] Docker. Docker Hub. https://hub.docker.com.

[14] Docker. Docker Security Scanning. https://docs.docker.com/v17.12/docker-cloud/builds/image-scan.

[15] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to support PaaS. In *Proceedings of the International Conference on Cloud Engineering*. IEEE, 2014.

[16] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 2017.

[17] Google. Containers at Google. https://cloud.google.com/containers.

[18] Hewlett Packard Enterprise. Netperf. https://github.com/HewlettPackard/netperf.

[19] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies*. ACM, 2018.

[20] Instana. Stan's Robot Shop - A Sample Microservice Application. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application.

[21] IO Visor Project. eXpress Data Path. https://www.iovisor.org/technology/xdp.

[22] IO Visor Project. extended Berkeley Packet Filter. https://www.iovisor.org/technology/ebpf.

[23] iPerf. Network bandwidth measurement tool. https://iperf.fr.

[24] Zhiqiang Jian and Long Chen. A Defense Method against Docker Escape Attack. In *Proceedings of the International Conference on Cryptography, Security and Privacy*. ACM, 2017.

[25] Amit Joshi, Andrew Leung, Corin Dwyer, Fabio Kung, Sargun Dhillon, Tomasz Bak, Andrew Spyker, and Tim Bozarth. Titus, the Netflix container management platform, is now open source. *Netflix Technology Blog*, 2018.

[26] Kubernetes. https://kubernetes.io.

[27] Linux Foundation cOLLABORATIVE Project. Open vSwitch. https://www.openvswitch.org.

[28] Mace. Docker OpenVPN container. https://hub.docker.com/r/mace/openvpn-as.

[29] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, 2017.

[30] MemSQL. Docker MemSQL container. https://hub.docker.com/_/memsq.

[31] Microservice Architecture. https://microservices.io/patterns/microservices.html.

[32] Amr A Mohallel, Julian M Bass, and Ali Dehghantaha. Experimenting with Docker: Linux Container and BaseOS Attack Surfaces. In *Proceedings of the International Conference on Information Society*. IEEE, 2016.

[33] NCCGroup. Abusing Privileged and Unprivileged Linux Containers. https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers.

[34] Netfilter and IPtables. https://www.netfilter.org.

[35] Nginx. Nginx Docker Container. https://hub.docker.com/_/nginx.

[36] RedHat. Atomic Scan - Container Vulnerability Detection. https://developers.redhat.com/blog/2016/05/02/introducing-atomic-scan-container-vulnerability-detection.

[37] RedHat. Benchmarking nftables. https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables.

[38] Redis. Redis Docker Container. https://hub.docker.com/_/redis.

[39] Romana. Romana v2.0. https://romana.io.

[40] Seccomp sandbox. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[41] SELinux Project. http://selinuxproject.org/page/Main_Page.

[42] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[43] Rui Shu, Xiaohui Gu, and William Enck. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Conference on Data and Application Security and Privacy*. ACM, 2017.

[44] StackRox. https://www.stackrox.com.

[45] StackRox. Breaking Bad: Detecting real world container exploits. https://www.stackrox.com/post/2018/03/breaking-bad-detecting-real-world-container-exploits.

[46] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of the Security Symposium*. USENIX, 2018.

[47] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding Security Implications of Using Containers in the Cloud. In *Proceedings of the Annual Technical Conference*. USENIX, 2017.

[48] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. Security Analysis of Container Images Using Cloud Analytics Framework. In *International Conference on Web Services*. Springer, 2018.

[49] Tigera. Project Calico. https://www.projectcalico.org.

[50] Tripwire. State of Container Security Report. https://www.tripwire.com/state-of-security/devops/organizations-container-security-incident.

[51] TwistLock. https://www.twistlock.com.

[52] TwistLock. A Busybox autocompletion vulnerability. https://www.twistlock.com/2017/11/20/cve-2017-16544-busybox-autocompletion-vulnerability.

[53] TwistLock. Escaping Docker container using waitid. https://www.twistlock.com/2017/12/27/escaping-docker-container-using-waitid-cve-2017-5123.

[54] Weaveworks. Sock Shop - A Microservices Demo Application. https://microservices-demo.github.io.

[55] Weaveworks. Weave Net. https://www.weave.works/oss/net.

[56] Yelp. How Yelp Runs Millions of Tests Every Day. https://engineeringblog.yelp.com/2017/04/how-yelp-runs-millions-of-tests-every-day.html.

[57] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. OpenNetVM: A platform for high performance network service chains. In *Proceedings of the workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 2016.

[58] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings in the Symposium on Networked Systems Design and Implementation*. USENIX, 2019.

Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure

[†‡]Shuai Xue, [†‡]Shang Zhao, [†‡]Quan Chen, [‡]Gang Deng, [‡]Zheng Liu, [‡]Jie Zhang, [‡]Zhuo Song
[‡]Tao Ma, [‡]Yong Yang, [‡]Yanbo Zhou, [‡]Keqiang Niu, [‡]Sijie Sun, [†]Minyi Guo
[†]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[‡]*Alibaba Cloud*

Abstract

Ensuring high reliability and availability of virtualized NVMe storage systems is crucial for large-scale clouds. However, previous I/O virtualization systems only focus on improving I/O performance and ignore the above challenges. To this end, we propose Spool, a reliable NVMe virtualization system. Spool has three key advantages: (1) It diagnoses the device failure type and only replaces the NVMe devices with actual media errors. Other data link errors are handled through resetting the device controller, minimizing data loss due to unnecessary device replacement. (2) It ensures the consistency and correctness of the data when resetting the controller and upgrading the storage virtualization system. (3) It greatly reduces the restart time of the NVMe virtualization system. The quick restart eliminates complaints from tenants due to denial-of-service during a system upgrade and failure recovery. Our evaluation shows that Spool provides reliable storage services with performance loss smaller than 3%, and it reduces restart time by 91% when compared with SPDK.

1 Introduction

In large-scale public clouds, the cores and memory are virtualized and shared by multiple tenants. A single physical server can serve up to 100 virtual machines (VMs) from either the same or different tenants [41]. On the physical server, VMs are managed with VM hypervisors, such as VMware [13], KVM [25], and Xen [14]. The hypervisors are also responsible for handling the interactions between the guest operating system in the VMs and the host operating system on the physical server.

Virtualizing I/O devices so that tenants can share them has attracted the attention of both industry and academia [15, 23, 31, 33, 40, 42]. A guest VM mainly stores and accesses its data on local devices through the I/O virtualization service with high throughput and low latency. For instance, the Big Three of cloud computing (Amazon EC2 I3 series [2], Azure Lsv2 series [3], and Alibaba ECS I2 series [1]) are providing the next generation of storage optimized instances for workloads that require high I/O throughput and low latency. These products are driven by local devices that eliminate the long latency over the network [8]. At the same time, accessing data from local devices increases the risk of a single point of failure as the reliability of data is dependent on the reliability of the host node.

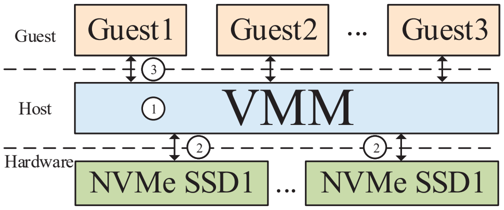


Figure 1: Virtualizing NVMe-based storage system.

Solid-state drives (SSDs) are often adopted as storage devices due to their high throughput and low latency compared to those of hard drives. In particular, the recent NVM Express (NVMe) interface [9] further increases the I/O performance of SSDs compared with the traditional SATA interface. Mainstream storage virtualization solutions, such as Virtio [29], support NVMe devices. Because serious performance degradation is observed in I/O virtualization [22], userspace NVMe driver in QEMU [43], Storage Performance Development Kit (SPDK) [38], SPDK vhost-NVMe [39], and Mdev [27] have been proposed to further improve the I/O throughput of virtualized NVMe devices.

While prior researchers focused on improving the read/write throughput and reducing the latency of virtualized NVMe devices, they ignored the reliability problem although it is equally important. In large-scale public clouds, NVMe device failures occur due to heavy use and the need for NVMe virtualization systems to be upgraded often to add new features or apply new security patches. Emerging NVMe virtualization systems fail to handle failure recovery and system upgrades efficiently. To better explain this problem, Figure 1 shows an example where multiple tenants share a virtualized NVMe storage system on a physical node.

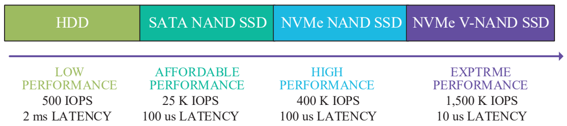


Figure 2: Development of the hardware I/O performance.

With emerging virtualized storage systems, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through either cold-plug or hot-plug. This failure recovery mechanism results in unnecessary data loss from the failed NVMe device. The statistics of our in-production cloud show that only 6% of 300,000 device failures involve media errors that can only be resolved by replacing with a new device. Other device failures are caused by data link errors that can be resolved by resetting the NVMe device controller. Resetting an NVMe device's controller would not result in data loss from the device, and we can perform the reset operation fast without removing the failed device (② in Figure 1) and restarting the virtualization system (① in Figure 1).

The standard procedure for upgrading the virtualized storage system on a node is stopping the daemon process that runs the system, updating the binary file, and then initializing the whole software stack of the virtualization system again [17]. In this period, all the I/O devices on the node are inaccessible due to the lack of an I/O virtualization system. Our measurement shows that the software initialization procedure already spends approximately 2.5 s probing all the I/O devices and SPDK's Environment Abstraction Layer (EAL) [38] (to be discussed in detail in Section 5). This long downtime hurts the user experience. A possible solution to reduce the impact of the upgrade is migrating the VMs (and the corresponding data) to other nodes [19, 41]. However, live VM migration is too costly for regular backend updates, especially when a large amount of backend requires updating, for example, when applying an urgent security patch.

However, the data written by the Guest VMs may be lost when resetting the controller or performing the upgrade (③ in Figure 1). The loss happens in the case that the data persisted in the NVMe device (still in the submit queue) when the reset operation or the process restart was performed. No prior work on NVMe virtualization has considered such a reliability problem.

To resolve the above problems, we propose *Spool*, a holistic reliable virtualized NVMe storage system for public clouds with local disks. Compared with prior NVMe virtualization systems, Spool has the following key advantages: (1) It diagnoses the device failure type and only replaces the NVMe devices with actual media errors. Other data link errors are handled through resetting the controller, minimizing data loss due to the unnecessary disk replacement. (2) It ensures the consistency and correctness of the data when resetting the controller and upgrading the virtualization system. (3) It greatly reduces the restart time of the NVMe virtualization system to approximately 100 milliseconds. The quick restart eliminates complaints from tenants due to denial-of-service during system upgrades and failure recovery.

To be more specific, Spool is comprised of a *cross-process journal for recovery*, an *isolation-based failure recovery component*, and a *fast restart component*. The cross-process journal resides in the shared memory and records the data status from all the VMs. Even if Spool is restarted, the data in the journal is accessible for the new Spool process. Furthermore, an instruction merge is proposed to eliminate the inconsistency of the journal with minimal overhead. An "instruction" is a step within a transaction that updates the journal. The failure recovery component diagnoses the NVMe device error. Based on the error code, Spool either isolates and replaces the devices that have media errors or resets the controller (using the journal for reliability). The restart component records the runtime data structures of the current Spool process in the cross-process journal. By reusing the data structures at the restart for a system upgrade, we significantly reduce the downtime.

To the best of our knowledge, Spool is the first holistic virtualized system that is capable of handling hardware failure and NVMe virtualization system upgrades reliably. Spool is currently deployed in an in-production cloud that includes more than 20,000 physical nodes and 200,000 NVMe-based SSDs.

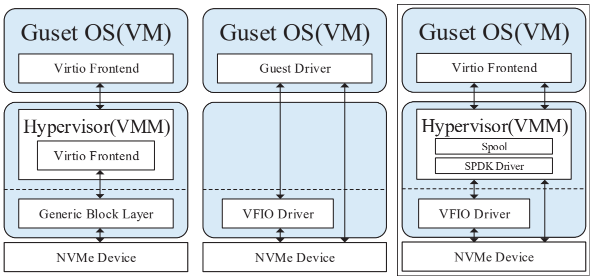
The main contributions of this paper are as follows.

- **An instruction merge-based reliability mechanism.** The instruction merge eliminates data inconsistent with the cross-process journal for recovery even if abnormal exits occur.
- **A restart optimization method.** The method greatly reduces the downtime of Spool during the upgrade and enables frequent system upgrades for adding new features and applying patches without affecting the tenants.
- **A hardware fault processing mechanism.** The mechanism diagnoses the device failure types and only replaces the NVMe devices with media errors, minimizing data loss due to unnecessary disk replacement.

Our experimental results show that Spool provides reliable storage services based on a shared memory journal with less than 3% performance loss, and it reduces the system restart time by 91% when compared to SPDK.

2 Background and Motivation

In this section, we introduce the virtualized NVMe storage systems and the motivation behind the design of Spool.



(a) Virtio (b) Passthrough based on VFIO (c) Spool based on SPDK

Figure 3: Comparison of NVMe virtualization mechanisms.

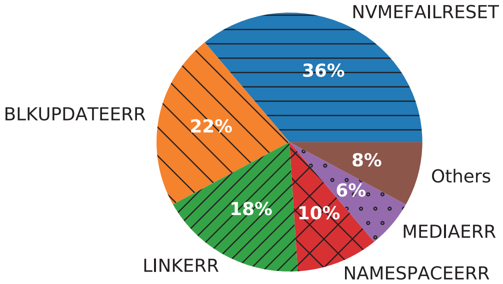


Figure 4: Breakdown of NVMe hardware failures.

2.1 Virtualized NVMe Storage Systems

The performance of an I/O device is impacted by both the storage media and the I/O software stack. As shown in Figure 2, Samsung NVMe SSD devices based on the latest V-NAND technology have increased the IOPS to 1.5 million and reduced the latency to 10 microseconds. In this scenario, the traditional SATA (Serial ATA) [34] interface for storage devices has become the performance bottleneck for such SSDs. Due to the limitation of the Advanced Host Controller Interface (AHCI) architectural design, the theoretical data transmission speed of the SATA interface is only 600 MB/s [34]. To solve the I/O bottleneck brought by the interface, the NVMe (Non-Volatile Memory Express) protocol [9] is designed and developed using a PCIe interface instead of SATA. Currently, NVMe supports deep queues with up to 64K commands to devices within a single I/O queue [9].

In public clouds, instead of selling raw hardware infrastructure, cloud vendors typically offer virtualized infrastructure as a service to maximize hardware resource utilization [16, 18]. Virtualization technology has shown its heroism, especially in the birth of hardware virtualization technology, such as Intel VT technology, which has greatly expanded the application scope of virtualization technology. There are three parts to the realization of virtualization: *CPU virtualization, memory virtualization*, and *I/O virtualization*. Among them, I/O virtualization requires more focus, and its performance directly determines the performance of the guest VM [22, 30, 32].

There are generally three I/O virtualization mechanisms: Virtio [29], VFIO [36], and SPDK-based userspace applications [38]. Figure 3 shows a comparison between Virtio, VFIO, and our SPDK-based design, Spool.

As for Virtio, the frontend exists in a guest OS, while the backend is implemented in a hypervisor, such as QEMU [12]. The frontend transfers I/O requests to the backend through the virtqueue, implemented as ring buffers, including *available ring* and *used ring* buffers. Available ring buffers could save multiple I/O requests driven by the frontend and transfer them to the backend for batch processing, which can improve the efficiency of information exchange between the client and hypervisor. However, a problem remains that each I/O request passes through the I/O stack twice for guest and host, whereas in modern storage devices based on NAND flash, the throughput and latency of VMs can only achieve 50% of the native performance [27].

As for VFIO, VMs directly access an NVMe device through passthrough, relying on hardware support (e.g., Intel VT-d). A VM approaches near-native performance on both latency and throughput with passthrough. However, a single device can only be assigned to one guest client. On the contrary, a host often runs multiple clients in a virtualized environment. It is difficult to ensure that each client can get a directly assigned device. Also, a large number of devices are allocated to clients independently, increasing the number of hardware devices as well as the cost of the hardware investment.

The Storage Performance Development Kit (SPDK) provides a set of tools and libraries for writing high-performance, scalable, user-mode storage applications. The bedrock of SPDK is a userspace, polled-mode, asynchronous, lockless NVMe driver [38]. SPDK enables zero-copy, highly parallel access direct to SSDs from a userspace application. User-mode drivers help improve the stability of the host operating system because they can only access the address space of the processes running them, and a buggy implementation does not cause system-wide problems. Spool is proposed based on the SPDK NVMe driver but focuses on the reliability of the virtualized storage system.

2.2 Reliability Problems

All the above I/O virtualization mechanisms ignore the high availability and reliability problems, although they are equally important in public clouds. To be more specific, state-of-the-art SPDK-based applications result in *unnecessary data loss* and *poor availability* when dealing with failed NVMe devices and upgrading applications, respectively.

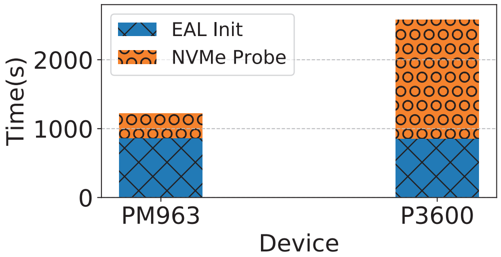


Figure 5: Breakdown of SPDK's start time on two NVMe SSDs.

2.2.1 Unnecessary Data Loss

If an NVMe device failure is detected on the hardware node, the device is in the failed state. When a device failure occurs on a node, all the VMs on the node are de-allocated and migrated to a healthy node by a standard procedure [11, 41]. After that, all the data on the failing node are securely erased. The victim tenants' data are lost and the tenants must proactively load their data on the new node again. With emerging virtualized storage systems like SPDK, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through hot-plug.

The above method results in significant unnecessary data loss because a single NVMe device may store data from multiple tenants, and NVMe devices have higher storage density, more vulnerable components (e.g., a Flash Translation Layer), and relatively higher failure rates. To demonstrate this problem in detail, we collected 300,000 NVMe device failures in our in-production environment. Figure 4 shows the breakdown of device failures. Most of the failures, 36%, are due to the NVMe controller failure error (NVMEFAILRESET). BLKUPDATEERR is the block update error. LINKERR is the PCIe interconnect link error. NAMESPACEERR is the NVMe device's namespace error. The pie chart shows that only 6% of the hardware failures are due to real media errors (MEDIAERR). Our investigation shows that most failures are caused by errors in the data link layer (e.g., namespace error, hardware link error, NVMe reset fail error), and these failures can be resolved by simply resetting the NVMe controller.

In summary, the current failure recovery method with SPDK results in significant unnecessary data loss.

2.2.2 Poor Availability

I/O virtualization systems tend to be upgraded frequently to add new features or apply security patches. When upgrading an I/O virtualization system, the key requirement is minimizing the I/O service downtime while ensuring the correctness of the data. There are two methods available to cloud vendors: VM live migration and live upgrade. Unfortunately, VM live migration is too costly for regular backend updates, especially when a large amount of backend requires updating, for example, when applying an urgent security patch, and for storing optimized instances with local NVMe storage , VM live migration is not even supported by cloud vendors [11]. The only way for us is to support the live upgrade and eliminate the downtime as much as possible.

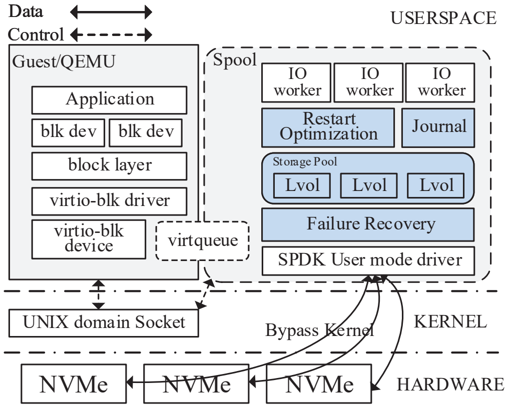


Figure 6: Design of Spool.

The I/O virtualization system must be restarted to complete the upgrade. With SPDK, we need to initialize the DPDK EAL library, probe the NVMe devices, and initialize the internal data structure of SPDK itself. SPDK spends a different amount of time in the "probe devices" step when resetting the controllers of different devices. The time required for each step is shown in Figure 5. As can be observed from this figure, the service downtime caused by the live upgrade is up to 1,200 ms for Samsung PM963 SSD. For Intel P3600, the total service downtime will be longer and lasts up to 2,500 ms.

In summary, the long downtime hurts the availability of the I/O virtualization system.

2.3 Design Principle of Spool

To resolve the unnecessary data loss and poor availability problems, we propose **Spool**, a holistic NVMe virtualization system. Spool is designed based on three principles:

- It should be able to identify the causes of device failure and adopt different methods to handle each failure. In this way, Spool eliminates most unnecessary NVMe device replacement.
- It should be able to optimize the restart procedure during a live upgrade so that the downtime can be minimized.
- It should be able to ensure that data access requests from the guest OS are not lost during a controller reset and live system upgrade.

3 Methodology of SPOOL

Figure 6 shows the design architecture of Spool, where the blue components are new relative to SPDK. Based on Spool, the NVMe devices on a node are virtualized and organized into a *Storage Pool (hence, "Spool")*. The virtualized NVMe devices are divided into multiple logical volumes that are managed through the buddy system [28]. The logical volumes are exposed to the guest OS in the form of block devices. As shown in the figure, the guest drivers communicate with Spool over shared memory. Specifically, the I/O worker on the host node polls I/O requests from the vhost virtqueue of block devices and submits to the corresponding physical devices. Spool is comprised of a *cross-process journal*, an *isolation-based failure recovery component*, and a *fast restart component*. Based on the three components, Spool ensures high reliability and availability of the storage pool.

The cross-process journal records each I/O request and its status to avoid data loss. The journal provides data access across process lifecycles, even if Spool restarts for an upgrade or exits abnormally. An instruction merge mechanism is proposed to eliminate the possible inconsistency of the journal itself due to an abnormal exit and to avoid the copy overhead of atomic operations.

The restart component records the runtime data structures of the current Spool process in shared memory. Spool catches the termination signals including *SIGTERM* and *SIGINT* to ensure the completion of all INFLIGHT I/O requests before actual exit. Spool reuses the data structures at the restart, thus significantly reducing the downtime spent on initializing the Environment Abstraction Layer (EAL) and resetting the device controller.

Spool diagnoses the device failure type online through self-monitoring, analysis, and reporting technology (S.M.A.R.T.) data [35]. For media errors, the failure recovery component isolates the failed device so that the administrator can replace the failed device through hot-plug. All the other NVMe devices are unaffected by the failed device. For data link errors, the recovery component resets the device's controller directly, thus minimizing data loss due to unnecessary disk replacement.

We implement Spool based on the SPDK userspace NVMe driver. Spool combines the advantages of Virtio and VFIO (Figure 3). Furthermore, instead of implementing the actual Virtio datapath, we offload the datapath from QEMU to Spool adopting the vhost-user protocol. Adopting this protocol, the guest OS directly interacts with Spool without QEMU's intervention. In addition, by adopting the SPDK userspace polled driver specification [38], Spool eliminates the overhead of system calls and data copies between kernel space and userspace stacks on the host and achieves high I/O performance.

4 Reliable Cross-Process Journal

In this section, we describe the reliability problem in the Virtio protocol that virtualizes the NVMe device, and we present the design of a cross-process journal that improves reliability.

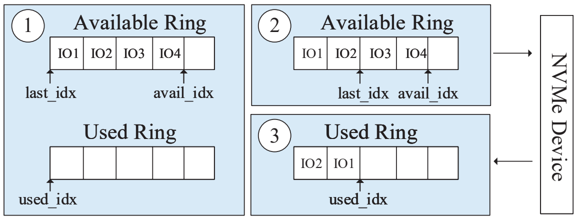


Figure 7: Design of Virtio block virtualization protocol.

4.1 Problem Statements

Figure 7 shows the design of the Virtio block driver that handles I/O requests in the guest OS. The I/O requests are processed in a *producer-consumer* model, where the guests are producers and the storage virtualization system is the consumer.

Specifically, the Virtio driver of each guest OS maintains an available ring and a used ring to manage its I/O requests. ① When an I/O request is submitted, the descriptor chain of the request is placed into the descriptor table. The descriptor chain includes the metadata, buffer, and status of the request. The metadata indicates the request type, request priority, and the offset of read or write. The guest driver places the index of the head of the descriptor chain into the next ring entry of the available ring, and the available index of the available ring ("`avail_idx`" in Figure 7) is increased. Then, the driver notifies the storage virtualization system that there is a pending I/O request. ② Meanwhile, the storage virtualization system running in the host obtains the several head indexes of the pending I/O requests in the available ring, increases the last index of the available ring ("`last_idx`" in Figure 7), and submits the I/O requests to NVMe device hardware driver. ③ Once a request is completed, the storage virtualization system places the head index of the completed request in the used ring and notifies the guest. Here, it is worth noting that the available ring and used ring are allocated by the guest, and the `avail_idx` is maintained by the guest, while both the `last_idx` and `used_idx` are maintained by the storage virtualization system.

The storage virtualization system may adopt either interrupt or polling to obtain I/O requests from the guest OS. Polling is able to fully utilize the advantages of NVMe devices to reap significant performance benefits [27, 37, 38], and Spool uses a dedicated I/O thread to poll I/O requests from the guest and data from the NVMe device instead of interrupts. This mechanism is implemented based on the SPDK userspace NVMe driver.

In general, the storage virtualization system runs well with the above procedure. However, if the storage virtualization system restarts for an upgrade or the NVMe device controller is reset, data loss may occur.

In the case of Figure 7, the storage virtualization system obtains two I/O requests, *IO1* and *IO2*. Then, the `last_idx` is incremented from *IO1* to *IO3* in the available ring. If the storage virtualization system restarts at this moment, the last available index will be lost, which means that it does not know where to proceed with I/O requests after the restart. Even if the last available index persists, there is no way to know whether the obtained *IO1* and *IO2* have been completed. If we simply continue to process the next request based on the last available index, the previously obtained incomplete request will be lost.

When we reset the controller of an NVMe device, all the admin and I/O queue pairs are cleared. Suppose that *IO1* and *IO2* have been submitted but they are still in the device I/O queue and have not been processed. Due to the lack of an I/O request state, the submitted I/O request in the cleared I/O queue pairs will never be checked as completion from the NVMe device.

In summary, a journal is needed to maintain I/O consistency.

4.2 Design of the Journal

We propose a cross-process journal of data that persists in shared memory to solve the problem of data loss caused by the storage virtualization system restart or device controller reset. Spool persists the following data in the journal.

- Last available index (`last_idx`) of the available ring. The index references to the starting descriptor index of the latest request that the Virtio backend reads from the available ring.

- The head index of each request in the available ring. The index refers to the head of a descriptor chain in the descriptor table.

With the `last_idx` in the journal, Spool knows which requests have been processed after restarting for the upgrade and is able to continue processing the remaining to-be-processed requests. If a request's starting descriptor index is referenced between `last_idx` and `avail_idx` of the available ring, it is a to-be-processed request.

Algorithm 1 Algorithm of cross-process journal

```
Require: head1: The head index of request in the available ring;
Require: head2: The head index of completed request from the driver;
Require: req[]: A ring queue to mark each I/O reqeust in the journal;
Require: aux: A temporary union variable to record multiple variables;
 1: poll head1 from the available ring;
 2: aux.state = START;
 3: aux.last_idx = journal->last_idx+1;
 4: aux.last_req_head = head1;
 5: *(volatile uint64_t *)&journal->val = *(volatile uint64_t *)&aux.val;
 6: req[head1] = INFLIGHT;
 7: journal->state = FINISHED;
 8: submit I/O request to driver;
 9: poll head2 completion;
10: journal->usd_idx++;
11: req[head2] = DONE;
12: put head2 to the used ring, may goto 10 or 13;
13: update used_index of the used vring with usd_idx of journal;
14: req[head2] = NONE;
```

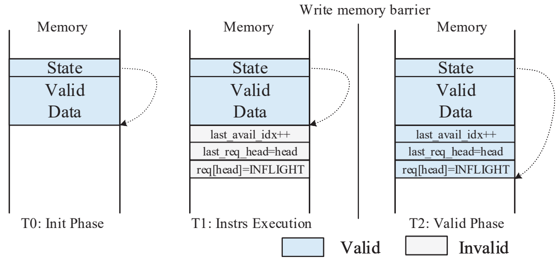


Figure 8: Transactional execution of multiple memory access instructions.

At the same time, when processing an I/O request in Spool, the request is given one of three states: INFLIGHT, DONE, or NONE. The cross-process journal uses Algorithm 1 to manage the I/O requests. To be more specific, when Spool gets a request from the frontend, it persists the head index of this request and marks the request as INFLIGHT, updates `last_idx`, and submits the request to the hardware driver. Once the I/O request completes, Spool updates the persisted `used_idx` in the journal and marks the request as DONE. After that, Spool returns the result of this request to the frontend, updates the used index of the frontend, and marks the request as NONE.

Adopting this method, if Spool restarts, the new Spool process can find out which request was not completed before the restart. In this way, the new Spool process resubmits the requests in the INFLIGHT state.

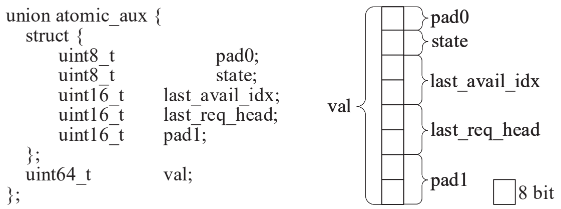
4.3 Merging Journal Update Instructions

An intuitive idea is to use shared memory as a journal to save this information with low latency overhead. However, it is challenging to ensure the consistency of the journal itself because Spool must update the journal multiple times during the processing of an I/O request.

Specifically, the process of each I/O request in Spool involves updating the last available index and marking the state of the request as INFLIGHT. During the processing, if Spool restarts or the controller is reset between the first two instructions, this request is lost.

If we can guarantee that instruction 3 (increase `last_idx`) and instruction 6 (change the request's status) in Algorithm 1 are executed in an atomic manner, the request loss problem can be resolved. However, only reading or writing a quadword aligned on a 64-bit boundary is guaranteed to be carried out atomically [7] in the memory, and the two instructions are not atomic when operating on the cross-process journal that resides in the memory.

To resolve the above problem, we design a multiple-instruction transaction model to guarantee atomic execution of the two instructions. As shown in Figure 8, each transaction consists of three phases. In T0, the init phase, we make a copy of the variable to be modified, such as `last_idx`, and in T1,



```
union atomic_aux {
    struct {
        uint8_t          pad0;
        uint8_t          state;
        uint16_t    last_avail_idx;
        uint16_t    last_req_head;
        uint16_t    pad1;
    };
    uint64_t        val;
};
```

Figure 9: Aux data structure that enables the update of multiple indexes using a single instruction.

the transaction will be in the START state. After all the instructions complete, the transaction will be in the FINISHED state in T2. Because the state is guaranteed to be carried out atomically, once the last available index updates, the related request is recorded as INFLIGHT in the journal. If any failure occurs in one transaction, we rollback the transaction to erase all data modifications with the copy.

As shown in Figure 9, we also design an auxiliary data structure carefully to eliminate the overhead of making a copy in T0 using a union type. The state, last available index, and head index of the related request are padding to 64 bits and a union memory block with a 64-bit value. We could update these three records within one instruction in Algorithm 1 step 5. This is a valuable trick to efficiently maintain journal consistency.

4.4 Recovering I/O Requests from Journal

Spool uses Algorithm 2 to recover the unprocessed I/O requests before the restart. With multiple journal transactions and an auxiliary structure, the new Spool process before the restart only needs to check the state and decide whether to redo the transactions or not.

Algorithm 2 Algorithm for recovering I/O requests

```
if (state == START) {
    req[last_get_req_head] = INFLIGHT;
    state = FINISHED;
}
jstate = (last_used_idx == used_idx) ? NONE : INFLIGHT
change all requests with done status to jstate;

last_used_idx = used_idx;
submit all requests marked as INFLIGHT;
```

The recovery algorithm works based on the value of the used index of vring and the last used index in the journal. If they are equal, Spool may crash after step 13 in Algorithm 1, but we do not know whether step 14 completes. Therefore, Spool tries to execute step 14 again and changes the states of the DONE requests to NONE. Otherwise, the request's process may be broken between steps 10 and 12. In this case, we do not know whether the request in the state DONE has been submitted to the frontend. To avoid losing any I/O request, we roll back the status of all the DONE requests to INFLIGHT. Because the frontend always has correct data, we synchronize the last used index in the journal with the frontend used index. In the last step, Spool resubmits all the requests that are in the INFLIGHT state.

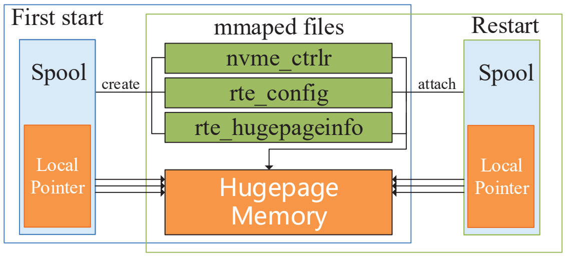


Figure 10: Boosting the restart of Spool by reusing the stable configurations.

Now, in Spool, the size of a single journal is only 368 bytes because it only records the metadata and the indexes of the requests in the available ring. Note that the above algorithm does not take precautions against the journal wrapping around; this is not possible because the journal is the same size as the available ring, so the guest driver will prevent such a condition.

5 Optimizing Spool Restart

As shown in Figure 5, when restarting a storage virtualization system, it initializes the EAL (Environment Abstraction Layer) in the DPDK driver and probes the NVMe devices in the SPDK driver on the host node. The relatively long restart time (ranging from 450 ms to 2,500 ms) hurts the availability as the whole storage system is out of service before the restart completes. In this section, we describe the method of optimizing the restart procedure in Spool.

5.1 Reusing Stable Configurations

During EAL initialization, the DPDK driver reserves all the memory in an IOVA-contiguous manner and sets up the huge pages for memory usage, such as I/O request buffers that are shared by the host userspace datapath and the physical device to perform DMA transactions. To be more specific, DPDK maps all the available huge pages to the process address space of DPDK, reads "`/proc/self/pagemap`" from the host OS to find the corresponding physical addresses of the huge pages, and sorts and merges the physical addresses into large contiguous chunks. After that, DPDK uses the physically continuous huge page chunks as memory segments. This design choice enables better hardware data prefetching and results in higher communication speed. Our experiment shows that the whole time spent on obtaining the physical memory layout information of huge pages is approximately 800 milliseconds, accounting for 70.9% of the total down time for a Samsung PM963 NVMe device. In our cloud, all the SSDs are restarted by more than 800,000 times in total in a single year. More restarts are required to periodically update the SPDK driver or apply new security patches. The long EAL initialization results in a long restart time and poor tenant experience.

Based on the above findings, we optimize the initialization steps of Spool. Specifically, the new Spool process after restart reuses the memory layout information from the current Spool process. Figure 10 shows the way we enable memory layout reuse. As shown in the figure, after the first startup of Spool, we store the related information (e.g., the huge pages in use and the virtual addresses of the huge pages) in the memory-mapped files that reside in the memory. If Spool restarts, it directly obtains the required information from the memory-mapped files in the memory with short latency and operates normally. Specifically, the "`rte_config`" file stores the global runtime configurations, and the "`ret_hugepageinfo`" file stores the memory layout information related to the huge page chunks used by Spool.

The above design does not guarantee that the new Spool after the restart will still use the largest continuous physical memory. This is because other processes may release huge pages and form larger continuous physical memory chunks. This design choice eliminates the long scan time of huge pages and does not degrade the performance of Spool.

5.2 Skipping Controller Reset

When probing the NVMe devices during the restart of the SPDK driver, more than 90% of the time is spent resetting the controller of NVMe devices. On an Intel P3600 SSD, the NVMe probe stage takes more than 1500 milliseconds (Figure 5). During the reset of the controller, SPDK frees the current admin queue, the I/O queue[1] in the controller, and creates them again for the controller after the reset.

Compared with SPDK, Spool skips the controller reset step during restart and reuses the data structures of the controller. This design is valid because the restart of Spool is not caused by media errors or data link errors. In this case, the data structures of the NVMe device controllers are not broken. To achieve reuse, Spool saves the NVMe device controller-related information in the memory-mapped file "`nvme_ctrlr`", as shown in Figure 10). After Spool restarts, it reuses the data of the device controller.

The challenging part here is that the context of the I/O request has disappeared with the exit of Spool. Therefore, we need to ensure the admin queue and I/O queue are completely clean.

In general, various signals are used to terminate one running process for an OS, such as *SIGTERM*, *SIGINT*, and *SIGKILL*. The default action for all of these signals is to cause the process to terminate. To gracefully terminate, we catch the termination signals including *SIGTERM* and *SIGINT* to ensure the completion of all INFLIGHT I/O requests before actually terminating. In that case, we could skip the reset operation after restart. Regarding the *SIGKILL* signal, which could not be handled in process or abnormal exit, we reset the controller after restart as usual.

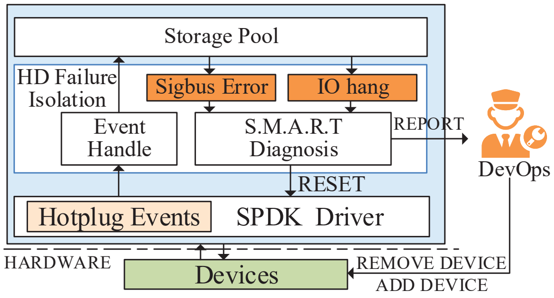


Figure 11: Handling hardware failures in different ways.

6 Hardware Fault Diagnosis and Processing

Traditionally, any NVMe device hardware failure causes the whole machine to be offline and repaired, and all VMs on the failing node need to be proactively migrated to a healthy node. With emerging virtualized storage systems like SPDK, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through hot-plug. On one hand, this causes data loss for users, and on the other hand, it increases operating costs.

To minimize data loss and reduce operating costs, we have implemented a fault diagnosis to identify the type of hardware fault and effectively avoid unnecessary hardware replacement.

6.1 Handling Hardware Failures

In large-scale cloud-based productions, hardware failures are frequent and may cause *SIGBUS* error and I/O hang, as shown in Figure 11.

Spool adopts the SPDK userspace NVMe driver to access a local NVMe PCIe SSD. Base address register (BAR) space for the NVMe SSD will be mapped into the user process through VFIO, which allows the driver to perform MMIO directly. The BAR space will be accessed by Spool while guest VMs send I/Os to the devices. However, the BAR space may become invalid while the device fails or is hot-removed. At this time, it will trigger *SIGBUS* error and cause the host process to crash if guest VMs still send I/O requests to the failed device of the host.

To improve reliability, a *SIGBUS* handler is registered into Spool. Once guest VMs send I/O requests to failed devices and access illegal BAR space, the handler will capture the *SIGBUS* error and remap the invalid BAR space to a dummy virtual address so that the *SIGBUS* error will not be triggered

[1] There are two types of commands in NVMe: Admin Commands are sent to Admin Submission and Completion Queue. I/O Commands are sent to I/O Submission and Completion Queues [9].

Table 1: Experimental configuration

Host configuration	
CPU & Memory	2x E5-2682v4 @2.5GHz; 128GB DDR4 Memory
NVMe devices	2 Samsung PM963 3.84TB SSDs
OS info	CentOS 7 (kernel verison 3.10.327)
Guest OS configuration	
CPU & Memory	4 vCPU; 8 GB
OS info	CentOS 7 (kernel verison 3.10.327)

again. Then, it sets the NVMe controller state to failure and fails all the internal requests in NVMe qpairs of the failed device.

6.2 Failure Model

The S.M.A.R.T. diagnosis collects and analyzes S.M.A.R.T. data to identify the failure type. S.M.A.R.T. data is disk-level sensor data provided by the firmware of the disk driver, including smart-log, expanded smart-log, and error-log, which can be used to analyze internal SSD errors.

Once a hardware media error is verified, Spool proactively fails the submitted I/O requests and returns I/O errors. After that, all the subsequent I/O requests to the failed device will return errors to guest VMs directly. Meanwhile, the S.M.A.R.T. diagnosis will send a report to DevOps. The hot-plug feature in the driver layer of SPDK is utilized in Spool; hence, the failed device can be replaced directly.

For the other hardware errors, such as a data link layer failure, diagnosis informs Spool to reset the controller. During the reset process, the I/O requests from the guest VMs hang. After the device is fixed, the INFLIGHT requests in the journal are resubmitted automatically.

7 Evaluation of Spool

In this section, we evaluate the performance of Spool in resolving hardware failure and supporting live upgrades. We first describe the experimental setup. Then, we evaluate the reliability of Spool in fixing an NVMe device's hardware failure without affecting other devices and correctly tolerating system upgrades at random times. After that, we show the effectiveness of Spool in reducing the system restart time, followed by a discussion on the impact of Spool on the I/O performance and the extra overhead caused by the cross-process journal. Lastly, we discuss the effectiveness of Spool on an in-production large-scale public cloud.

7.1 Experimental Setup

We evaluated Spool on a server equipped with two Intel Xeon E5-2682 processors operating at 2.5 GHz with 128 GB memory. For the NVMe devices, we adopted a mainstream SSD device: Samsung PM963 NVMe SSD. Table 1 summarizes the hardware and software specifications of the experimental platform.

For extensive evaluation, we use the Flexible I/O tester (FIO) [6] as our performance and reliability evaluation benchmark. FIO is a typical I/O tool meant to be used both for benchmark and stress/hardware verification and is widely used in research and industry. When evaluating the I/O performance, we use different parameters, as shown in Table 2, to demonstrate metrics, including IOPS and average latency recommended by Intel [5] and Alibaba [4]. To emulate the real-system cloud scenario, we split each NVMe SSD into three partitions (each partition was 100 GB) and allocated each partition to an individual VM.

We used libaio (Linux-native asynchronous I/O facility) [21] as the FIO load generation engine. Table 2 lists the generated FIO test cases. To obtain accurate performance, we tested raw SSDs without any file system.

Table 2: FIO test cases

Tested metrics	Test cases	FIO Configuration (bs, rw, iodepth, numjobs)
Bandwidth	Read	(128K, read, 128, 1)
	Write	(128K, write, 128, 1)
IOPS	Randread	(4K, randread, 32, 4)
	Mixread	(4K, randread 70%, 32, 4)
	Mixwrite	(4K, randwrite 30%, 32, 4)
	Randwrite	(4K, randwrite, 32, 4)
Average Latency	Randread	(4K, randread, 1, 1)
	Randwrite	(4K, randwrite, 1, 1)
	Read	(4K, read, 1, 1)
	Write	(4K, write, 1, 1)

7.2 Reliability of Handling Hardware Failure

When an NVMe device suffers from hardware failure, Spool isolates the failed device and performs device replacement or controller reset accordingly. When handling such failure, Spool should not affect the I/O operations on other devices of the same node, and the VMs that are using the failed device should receive I/O errors instead of exiting abnormally.

We designed an experiment to evaluate Spool in the above scenario. In the experiment, we launched two VMs on a hardware node equipped with two NVMe devices and configured the two VMs to uses different NVMe devices in Spool. The two VMs randomly read data from NVMe devices in the beginning, and we manually removed an NVMe device and observed the behavior of the two VMs.

Figure 12 presents the I/O performance of the two VMs when the NVMe device ("SSD2") was hot-removed at time 80 s. The hot remove was performed by writing a non-zero value to "`/sys/bus/pci/devices/.../remove`". Observed from this figure, the I/O performance of VM1 that uses the NVMe device SSD1 is not affected when SSD2 is removed. Meanwhile, VM2 does not exit abnormally after SSD2 is removed. Once a new SSD device replaces the failed SSD2 or the controller of SSD2 is reset correctly at time 95 s, VM2 is able to directly use SSD2 without any user interference.

Spool can successfully handle the above hardware failure because it catches the hardware hot-plug event and diagnoses the device failure type first. Hardware failures are handled

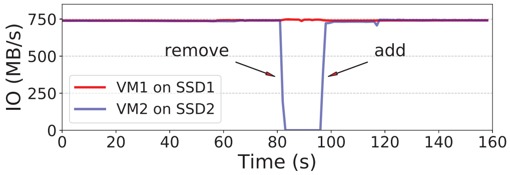


Figure 12: Handling hardware failure with Spool.

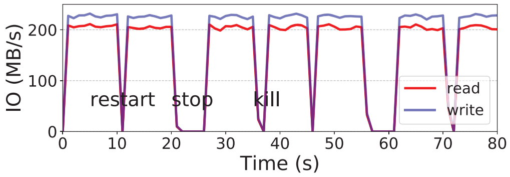


Figure 13: Data consistency at live upgrade with Spool.

in two ways: media errors are solved by hot-plugging a new SSD, and then the storage service automatically recovers, where the related logical devices are automatically mapped to new devices, while data link failure is handled by controller reset instead of replacing a SSD. On the contrary, if the traditional SPDK is used to manage SSDs, the hardware failure can only be solved by hot-plugging new devices, resulting in unnecessary data loss, and the storage service of SPDK needs to be reset manually for recovery.

7.3 Reliability of Handling Random Upgrades

To validate the reliability of Spool in handling upgrades without resulting in data loss, we designed an experiment that *restarts* purposely *stops and starts* and randomly *kills and restarts* Spool. In the experiment, we relied on the data verification function in FIO to check the data consistency. By enabling the data verification function, FIO verifies the file contents after writing 10 blocks contiguously with crc32 and reports whether any data corruption occurred or not. If FIO runs completely without errors, data consistency is verified.

Figure 13 shows the read and write performance to the SSD when we restart on purpose, stop and start, and randomly kill and restart Spool at time 10 s, 20 s, and 35 s. As can be observed from the figure, the I/O operations to the SSD complete correctly with Spool, even if Spool is directly killed and restarted for an upgrade.

Spool can guarantee data consistency during upgrades due to the cross-process journal. The journal persists the current states of all the NVMe devices. Whenever Spool is restarted, it is able to recover the states before the restart and continue to complete the unprocessed I/O requests. On the contrary, with SPDK, there is no mechanism to guarantee data consistency for INFLIGHT I/Os.

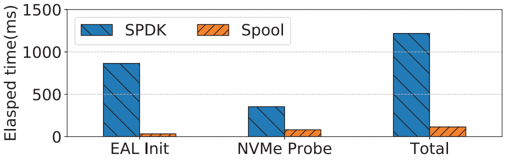


Figure 14: Restart times of Spool and SPDK.

7.4 Reducing Restart Time

Observed from Figure 13, the downtime due to the restart is short with Spool. In more detail, Figure 14 shows the restart time breakdown of Spool and SPDK.

As can be observed from this figure, Spool significantly reduces the total restart time from 1,218 ms to 115 ms on a Samsung PM963 SSD. This significant restart time reduction originates from the reduction of EAL initialization time and the NVMe probe time.

SPDK suffers from a long EAL initialization time and a long NVMe probe time because it initializes the EAL and resets the controller of the device during probing at each startup. By reusing the previous memory layout information, Spool minimizes the EAL initialization time. And, by skipping resetting the device controller, Spool reduces the NVMe probe time.

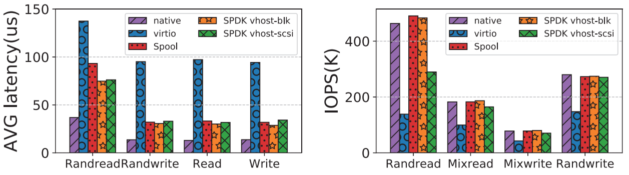
7.5 I/O Performance of Spool

It is crucial to ensure high I/O performance (i.e., high IOPS and low latency) when guaranteeing reliability. In this subsection, we report the I/O performance of NVMe devices with Spool in two cases: an NVMe device is only allocated to a single VM, and an NVMe device is shared by multiple VMs.

7.5.1 Case 1: Single VM Performance

Figure 15 presents the data access latency and IOPS to an SSD when it is virtualized with Virtio, SPDK, and Spool. In the figure, "native" shows the performance of the SSD measured on the host node directly; "SPDK vhost-blk" and "SPDK vhost-scsi" show the performance of the SSD if SPDK is used as the I/O virtualization system and the SSD is treated as a block device or a SCSI device, respectively.

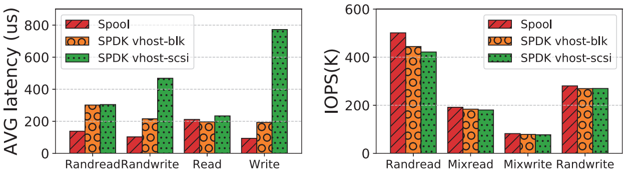
As can be observed from Figure 15, all the I/O virtualization systems result in longer data access latency compared with the native access due to extra layers in the virtualization system. Meanwhile, Spool achieves similar data access latency to SPDK. From the IOPS aspect, the IOPS of *Randread* with Spool is 2.54x higher than Virtio and even slightly better than the native bare metal. As mentioned in Section4, Spool uses polling instead of interrupt to monitor the I/O requests. Polling saves the expense of invoking the kernel interrupt handler and eliminates context switching. These results are consistent with prior work [27].



(a) Average latency

(b) IOPS

Figure 15: Average data access latency and IOPS of an NVMe SSD when it is used by a single VM.



(a) Average latency

(b) IOPS

Figure 16: Average data access latency and IOPS of an NVMe SSD when it is shared by multiple VMs.

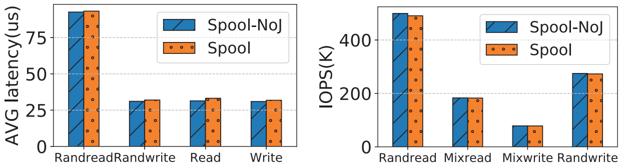
Compared with SPDK vhost-blk, the performance of our implementation is almost the same. Because the SPDK vhost-blk software stack is thinner than SPDK vhost-scsi, the IOPS with SPDK vhost-scsi is lower than that with Spool.

7.5.2 Case 2: Scaling to Multiple VMs

In this experiment, we partition an SSD into three logic disks and assign each logic disk to an individual VM. This experiment tests the effectiveness of Spool in handling multiple VMs on an NVMe device.

Figure 16 shows the IOPS and data access latency when three VMs share an NVMe device, and each result is the sum of those of all VMs. For the latency test, we ran each benchmark 10 times and report the average latency for each benchmark. As can be observed from this figure, Spool does not degrade the I/O performance of all the benchmarks compared with SPDK vhost-blk and SPDK vhost-scsi. Specifically, Spool improves the IOPS of *Randread* by 13% compared with SPDK vhost-blk, which reduces the average data access latency of *Randread* by 54% to 55% compared with SPDK vhost-blk and SPDK vhost-scsi, respectively.

Besides, we can see that the SSD device achieves similar IOPS when the SSD is used by a single VM and three VMs, and we can see the average data access by comparing Figure 15(b) and Figure 16(b). The data access latency of the benchmarks when the SSD is shared by three VMs is three times that of case 1. This is reasonable because the backend I/O load pressure increases linearly with the number of VMs, so the total latency of the three VMs increases. While the I/O load pressure of one VM has reached the throughput limit of the Samsung PM963 specification [10], the total IOPS of three VMs remains unchanged. The I/O performance of Spool is slightly better than that of SPDK because Spool and SPDK use different logical volume management mechanisms. Specifically, Spool uses the buddy system to manage logical volumes, while SPDK uses Blobstore.



(a) Average latency

(b) IOPS

Figure 17: Overhead of the cross-process journal.

7.6 Overhead of the Cross-Process Journal

To measure the overhead of the cross-process journal, we implement a Spool variation, Spool-NoJ that disables the cross-process journal. Figure 17 shows the data access latency and the IOPS of the SSD with Spool and Spool-NoJ.

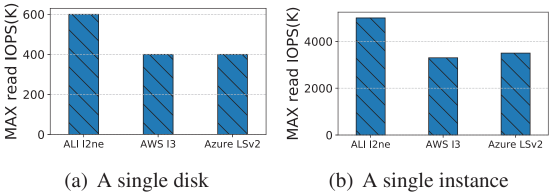
As shown in Figure 17, Spool-NoJ and Spool result in similar data access latency and IOPS. Compared with Spool-NoJ, Spool increases the average data access latency no more than 3%. Meanwhile, Spool reduces the IOPS by less than 0.76% compared with Spool-NoJ. The extra overhead caused by the cross-process journal in terms of average latency and IOPS throughput is negligible.

7.7 Deployment on an In-production Cloud

We currently deploy Spool in 210 clusters with approximately 20,000 physical machines equipped with approximately 200,000 NVMe SSDs.

On the cloud supported by Spool, we built a Platform-as-a-Service cloud(ALI I2 series) that provides low latency, high random IOPS, and high throughput I/O support. The maximum IOPS of single disk is 50% higher than that of competitive products and the maximum IOPS of a largest specification instance is 51% higher than that of the competitive products, as shown in Figure 18. The in-production cloud hosts Cassandra, MongoDB, Cloudera, and Redis. They are ideal for Big Data, SQL, NoSQL databases, data warehousing, and large transactional databases. Recently, the instance resources of local SSD disks have helped OceanBase break the world record for TPC-C benchmark performance test maintained by Oracle for 9 years and becoming an important milestone in the evolution history of global databases.

A holistic fine-grain monitoring system is crucial for clouds. While the monitoring system is able to diagnose media errors and other SSD failures, Spool handles the failures in different ways. Our statistics show that the current hardware failure rate is approximately 1.2% over a whole year. Throughout one year, approximately 2,400 out of 200,000 SSDs suffer from media errors. The media error is due to either media damage or life depletion. From the system upgrade aspect,



(a) A single disk (b) A single instance

Figure 18: Maximum read IOPS compared with AWS and Azure.

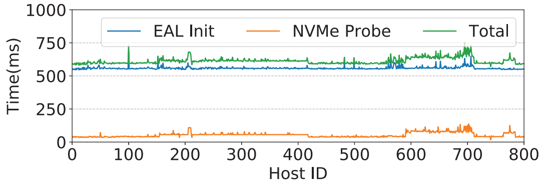


Figure 19: Restart time of Spool during a live upgrade.

we release a new version of Spool every six months. In total, we upgrade Spool on more than 40,000 physical machines every year. The purpose of the new version is to deal with two issues: 1) releasing new features and 2) fixing online stability during the operation and maintenance phase. Most of the new features are related to performance, such as adding support for multiple queues, optimizing memory DMA operation, and optimizing memory pools.

Figure 19 shows the restart time of some selected machines in a live upgrade in production. The *x*-axis is the ID of the physical node. Due to historical reasons, Spool in the production environment is based on the earlier version of the DPDK driver, and the initialization memory requires 1,172 ms, accounting for 70% of the initialization of EAL (1,792 ms in total), which is almost optimized. However, the rest of the initialization of the EAL is still 550 ms, and the total upgrade time for 95% of the machines is within 654 ms. We are working on updating the DPDK driver for Spool.

8 Related Work

There has been a lot of work concentrating on NVMe virtualization and optimizing storage I/O stacks for modern fast storage devices(e.g., NVMe SSDs).

Kim et al. [22] analyzed the overheads of random I/O over storage devices in detail and mainly focused on optimizing the I/O path by eliminating the overhead of user-level threads, bypassing the 4KB aligned I/O routine and enhancing the interrupt delivery delay in QEMU. In QEMU/KVM forum 2017, Zheng et al. [43] implemented a userspace NVMe driver in QEMU through VFIO to accelerate the virtio-blk in a guest OS at the cost of device sharing. Peng et al. [27] discussed the importance of polling for NVMe virtualization and took advantage of polling to achieve extreme I/O performance while each polling thread brings 100% usage of 3 cores.

Virtio [12] is a de facto standard for para-virtualized driver specifications including virtio-blk and virtio-scsi, which defines the common mechanisms for virtual device discovery and layouts. However, each I/O request passes through the I/O stack twice for guest and host, causing great loss of I/O performance. Then, a vhost acceleration method is proposed to accelerate virtio-scsi or virtio-blk provided by the storage performance development kit (SPDK) [38], such as kernel vhost-scsi, userspace vhost-scsi, and vhost-blk.

While local SSDs provide higher data access bandwidth and lower latency, storage disaggregation (e.g., ReFlex [24], NVMe-over-Fabrics [20], [23]) enables flexible scaling and high utilization of Flash capacity and IOPS at the cost of interconnecting latencies over a network. The bandwidth of NIC is the bottleneck to saturate the bandwidth of multiple disks. Two NVMe SSDs may saturate the bandwidth of emerging NIC (100GbE). Meanwhile, the upgrade of cloud services with minimal downtime has also been widely studied. Neamtiu et al. [26] highlighted challenges and opportunities for upgrades to the cloud. Clark et al. [19] proposed a live migration mechanism to temporarily move VMs to a backup server, upgrade the system, and then moves the VMs back. Zhang et al. [41] proposed Orthus to live upgrade the VMM without interrupting customer VMs and significantly reduce the total migration time and downtime. However, Orthus only focuses on KVM and QEMU and ignores the backend services.

9 Conclusion

This paper presented Spool, a holistic virtualized storage system that is capable of handling hardware failure and the NVMe virtualization system upgrades reliably. Spool significantly reduces the restart time by 91% on a Samsung PM963 SSD by reusing the data structures at the restart for system upgrades. Compared with emerging virtualized storage systems such as SPDK, Spool supports live upgrades and guarantees data consistency with a shared memory-based journal at any time. Moreover, Spool diagnoses device failure type instead of hot-plugging directly and eliminates most unnecessary NVMe device replacement.

Acknowledgement

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004800), the National Natural Science Foundation of China (NSFC) (61632017, 61772480, 61872240, 61832006, 61702328). Quan Chen and Minyi Guo are the corresponding authors. We thank Lingjun Zhu for his help in the experiments. And we also thank Mike Mesnier for shepherding our manuscript.

References

[1] Alibaba cloud instance family with local ssds. https://www.alibabacloud.com/help/doc-detail/25378.htm?spm=a2c63.p38356.879954.7.158f775aPotRZi#i2.

[2] Amazon ec2 i3 instances. https://aws.amazon.com/ec2/instance-types/i3/.

[3] Azure lsv2-series. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-storage#lsv2-series.

[4] Block storage performance. https://www.alibabacloud.com/help/doc-detail/25382.htm?spm=a2c63.p38356.879954.28.344791f30dgBQZ#title-1rp-8na-22y.

[5] Evaluate performance for storage performance development kit. https://software.intel.com/en-us/articles/evaluate-performance-for-storage-performance-development-kit-spdk-based-nvme-ssd.

[6] Fio. https://fio.readthedocs.io/en/latest/.

[7] Intel® 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[8] Local ssd. https://www.alibabacloud.com/help/doc-detail/63138.htm?spm=a2c63.p38356.879954.199.aa265e6dMUz7fg#concept-g3w-qzv-tdb.

[9] Nvm express specification. http://www.nvmexpress.org/specifications.

[10] Pm963 specifications. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZQLW3T8HMLP/.

[11] Utilizing local nvme storage on azure. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/storage-performance#utilizing-local-nvme-storage.

[12] Virtio homepage. https://www.linux-kvm.org/page/Virtio.

[13] Vmware homepage. https://www.vmware.com.

[14] Xen main page. https://wiki.xen.org/wiki/Main_Page.

[15] Amro Awad, Brett Kettering, and Yan Solihin. Non-volatile memory host controller interface performance analysis in high-performance i/o systems. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 145–154. IEEE, 2015.

[16] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[17] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44. ACM, 2006.

[18] Quan Chen and Qian-ni Deng. Cloud computing and its key techniques [j]. *Journal of Computer Applications*, 9(29):2562–2567, 2009.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[20] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–9, 2017.

[21] William K Josephson. An introduction to libaio. 2007.

[22] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. Improving i/o performance of nvme ssd on virtual machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1852–1857. ACM, 2016.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 29. ACM, 2016.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash $\approx$ local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.

[25] Uri Lublin, Yaniv Kamay, Dor Laor, and Anthony Liguori. Kvm: the linux virtual machine monitor. 2007.

[26] Iulian Neamtiu and Tudor Dumitraş. Cloud software upgrades: Challenges and opportunities. In *2011 International Workshop on the Maintenance and Evolution*

of Service-Oriented and Cloud-Based Systems, pages 1–10. IEEE, 2011.

[27] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: a nvme storage virtualization solution with mediated pass-through. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 665–676, 2018.

[28] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[29] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[30] Jeffrey Shafer. I/o virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, pages 5–5. USENIX Association, 2010.

[31] Sankaran Sivathanu, Ling Liu, Mei Yiduo, and Xing Pu. Storage management in virtualized cloud environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 204–211. IEEE, 2010.

[32] Yongseok Son, Hyuck Han, and Heon Young Yeom. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 8. ACM, 2015.

[33] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson, et al. {DC} express: Shortest latency protocol for reading phase change memory over {PCI} express. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 309–315, 2014.

[34] Wikipedia contributors. Serial ata — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Serial_ATA&oldid=932414849, 2019. [Online; accessed 26-December-2019].

[35] Wikipedia contributors. S.m.a.r.t. — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=S.M.A.R.T.&oldid=931348877, 2019. [Online; accessed 8-January-2020].

[36] Alex Williamson. Vfio: A user's perspective. In *KVM Forum*, 2012.

[37] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[38] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[39] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[40] Xiantao Zhang and Yaozu Dong. Optimizing xen vmm based on intel® virtualization technology. In *2008 International Conference on Internet Computing in Science and Engineering*, pages 367–374. IEEE, 2008.

[41] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. Fast and scalable vmm live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–105. ACM, 2019.

[42] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[43] Fam Zheng. Userspace nvme driver in qemu. In *KVM Forum 2017*, pages 25–27, 2017.

HDDse: Enabling High-Dimensional Disk State Embedding for Generic Failure Detection System of Heterogeneous Disks in Large Data Centers

Ji Zhang[§†], Ping Huang[§£], Ke Zhou[§*], Ming Xie[ζ], Sebastian Schelter[†]
[§]*Huazhong University of Science and Technology*
[£]*Temple University,* [ζ]*Tencent Inc.,* [†]*University of Amsterdam*
Ji Zhang and Ping Huang are the co-first authors

Abstract

The reliability of a storage system is crucial in large data centers. Hard disks are widely used as primary storage devices in modern data centers, where disk failures constantly happen. Disk failures could lead to a serious system interrupt or even permanent data loss. Many hard disk failure detection approaches have been proposed to solve this problem. However, existing approaches are not generic models for heterogeneous disks in large data centers, e.g, most of the approaches only consider datasets consisting of disks from the same manufacturer (and often of the same disk models). Moreover, some approaches achieve high detection performance in most cases but can not deliver satisfactory results when the datasets of a relatively small amount of disks or have new datasets which have not been seen during training. In this paper, we propose a novel generic disk failure detection approach for heterogeneous disks that can not only deliver a better detective performance but also have good detective adaptability to the disks which have not appeared in training, even when dealing with imbalanced or a relatively small amount of disk datasets. We employ a Long Short-Term Memory (LSTM) based siamese network that can learn the dynamically changed long-term behavior of disk healthy statues. Moreover, this structure can generate a unified and efficient high dimensional disk state embeddings for failure detection of heterogeneous disks. Our evaluation results on two real-world data centers confirm that the proposed system is effective and outperforms several state-of-the-art approaches. Furthermore, we have successfully applied the proposed system to improve the reliability of a data center and exhibit practical long-term availability.

1 Introduction

Device failure is a common problem in large data centers, where hard disks are widely used as the primary storage devices. A disk failure could lead to temporary data loss and thus system breakdown, or even permanent data loss if the lost data cannot be recovered by data protection schemes (eg., replication and erasure codes) due to disk failures exceeding the designed correction capability [1]. About 80% of system breakdowns are caused by hard drive failures in the data center [2]. Therefore, how to ensure the reliability of disks becomes an important issue in a storage system [3]. Although there are a series of passive fault defense mechanisms like EC (Erasure Codes) [4, 5] and RAID (Redundant Arrays of Independent Disks) [6], many researchers have focused on proactive disk failure detection which aims to ensure the reliability and availability of large-scale storage systems in advance [7–9]. Disk failure prediction is an important issue in system researches. Timely and accurate failure prediction can ensure the operational continuity of systems and even avoid data loss. It is because of this common realization among the storage research community, there have recently emerged a decent amount of researches on disk failure prediction.

We observed six classes of approaches of disk failure detection: threshold-based (TB) approaches [10, 11], distance-based anomaly detection (DAD) approaches [12–16], shallow machine learning (SML)-based approaches [17–29], deep neural network (DNN)-based approaches [30–34], one-class classification (OCC) based approaches [35–39] and transfer learning (TL)-based approaches [1, 40–44]. However, these approaches have their limitations as summarized in Table 1.

(1) Limited applicability. Different disk manufacturers have different S.M.A.R.T (Self-Monitoring, Analysis, and Reporting Technology) values or data distribution, even in different disk models of the same manufacturer [1, 43]. This will result in a situation where methods (SML, DNN, OCC, TL) trained on specific parameters only work well for the same manufacturer, and often even for the same models, limiting their scope of applicability in practice.

(2) Lack of adaptability. New disk models enter gradually to replace or augment the storage capacity, leading storage systems to consist of disks from different vendors and/or different models [1, 41, 42]. Facing these different types of heterogeneous disks, the existing prediction models (DAD, SML, DNN, OCC, TL) must be retrained to obtain more reliable predictions in order to adapt to the changes in the data distribution introduced by these new disk models. Such re-

Table 1: Characteristics of different approaches in disk failure detection. $(\star)$ refers to certain conditions that are required, e.g., finding a suitable source domain (i.e., another disk model) for knowledge transfer. All existing methods have certain disadvantages. We use TPR refers to True Positive Rate, FPR refers to False Positive Rate and F-Measure to evaluate these methods (for these evaluation metrics see Section 5.1.3).

	TB	**DAD**	**SML**	**DNN**	**OCC**	**TL**	**HDDse**
Applicability	√	√	×	×	×	×	√
Adaptability	√	√	×	×	×	√$(\star)$	√
Imbalance datasets	√	×	×	×	√	×	√
Minority Disk	√	×	×	×	×	√$(\star)$	√
Performance	TPR: 3%-10%	56%-70%	75%-96%	87%-98%	70%-92%	80%-97%	92%-97%
	FPR: 0%-2%	0%-1%	1%-4%	0%-1%	0%-10%	FPR: 0%-6%	0.2%-0.4%
	F-Measure: 2%-13%	49%-58%	67%-92%	86%-93%	65%-91%	77%-93%	91%-97%

training is tedious and expensive in a large data center.

(3) Imbalanced datasets. Imbalanced data refers to a situation where the number of samples is not the same for all the classes in a classification dataset. Most machine learning (ML) models tend to bias the class with the largest proportion of observations (known as majority class), which may lead to inaccuracies. This may be particularly problematic when we are interested in the correct classification of a "rare" class (also known as minority class). In real world cloud storage systems, the imbalanced ratio of positive (failure) samples and negative (healthy) samples poses a significant threat to the efficiency of machine learning models [2]. The most commonly used technique of existing methods (DAD, SML, DNN, TL) to approach this problem are under-sampling [1, 15, 28] and over-sampling techniques [45]. In under-sampling, the data samples are adjusted based on the class with the lowest sample count to keep the number of samples per class equal. In over-sampling the samples of the sparsely populated class are re-sampled to match the number of data samples in the other classes. Under-sampling discards a large amount of data while oversampling can easily cause overfitting in the model. Moreover, these approaches might increase the training cost.

(4) Minority disk failure detection. In large-scale storage systems, new disks gradually replace failed disks, which results in a situation where the data centers continuously contain small amounts of new disk models. Due to a lack of sufficient training data, some detective approaches (DAD, SML, DNN, OCC) fail to deliver satisfactory detective performance for these models. Although some TL methods (transfer learning is good at transferring knowledge from the source dataset to the target dataset) have been proposed to address this issue, their performance depends on finding a suitable source domain (in the form of another disk model) for knowledge transfer, which might be difficult in a real world data center (detailed in Section 3.1.2).

(5) Detective performance. The True Positive Rate (TPR), False Positive Rate (FPR) and F-Measure (all these evaluation metrics see Section 5.1.3 for details) are the commonly used metrics to measure the capabilities of classification models in disk failure detection [1, 15, 17, 19]. Most methods show unstable performance in practical long-term use and often fail to obtain both high TPR and low FPR.

In this paper, we propose a novel disk failure detection system called "High-Dimensional Disk State Embedding for Generic Failure Detection" (*HDDse*). It is a distance-based anomaly detection approach using deep learning and addresses the discussed shortcomings of the existing methods.

In summary, we provide the following contributions:

- To the best of our knowledge, we propose the first generic disk failure detection system *HDDse* for heterogeneous disks. It is not sensitive to imbalanced datasets, has wider applicability, well detects the minority disks and exhibits high adaptability. (Section 4.1)
- We propose a Long Short-Term Memory (LSTM)-based siamese network to calculate the similarity of disk health states in high-dimension space, which combines distance-based anomaly detection and deep learning to classify disk state in high-dimension. (Section 4.2)
- We experimentally evaluate our method on datasets from two real world data centers. We demonstrate that *HDDse* can effectively detect disk failures in long-term availability that greatly improves the reliability and availability of the storage system and outperforms the state-of-the-art approaches in five adopted metrics (Section 5.2 and 5.3).

2 Related Work

Existing work mostly uses the S.M.A.R.T (Self-Monitoring, Analysis and Reporting Technology) data to build a disk failure detection model. Almost all hard disk drives and flash-based SSDs now support S.M.A.R.T, which monitors the internal attributes of individual drives.

Threshold-based Methods (TB). All disk manufacturers use a thresholding algorithm which triggers a failure alarm when any single S.M.A.R.T attribute exceeds a predefined value [10, 11]. However, this approach provides only an estimated TPR (True Positive Rate, see Section 5.1.3 for details) of 3%-10% with a 0.1% FPR (False Positive Rate, see Section 5.1.3 for details). The reason is that the hard disk manufacturers set the thresholds conservatively to avoid expensive false alarm costs, i.e., they keep the FPR to a minimum at the expense of the TPR.

Distance-based Anomaly Detection Methods (DAD). DAD is the method of finding data objects with behaviors that are very different from expectation based on some similarity metrics. Shen et al. [15] utilize the change in the Euclidean distance [46] between the healthy disk samples and the last sample of a failed drive. Wang et al. [13] propose a model for drive anomaly prediction based on Mahalanobis distance (MD). In their subsequent study [14], they use a generalized likelihood ratio test over the dissimilarity vector to detect disk failures. Huang et al. [16] explore the read/write head failures and bad sector failures using the Euclidean distance to calculate the dissimilarity between each S.M.A.R.T record prior to the failure and the failure record. Gao et al. [12] present an incremental detection model of disk failures based on the Euclidean distance to measure the local anomalies of test points within their isolation regions optimizing the judgment of test point anomalies.

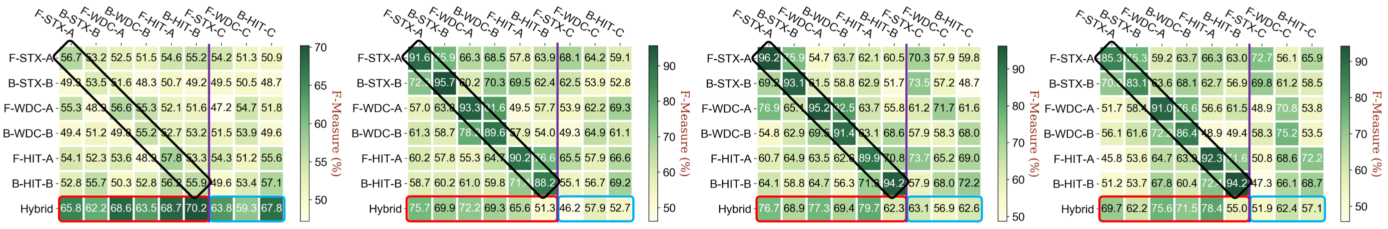
Shallow Machine Learning based Methods (SML). Conventional (shallow) machine learning methods typically require manual extraction and selection of features (eg., Support Vector Machines (SVM [47]), Logistic Regression (LR [48])), a critical step that is dispensed within the deep learning approach, i.e., it is automatic in deep learning approaches. Pitakrat et al. [29] evaluate and compare the performance of 21 machine learning algorithms for proactive disk failure detection. Li et al. [28] propose new drive failure detection models based on classification and regression trees. Yang et al. [27] design a disk failure prediction model based on L1-regularized logistic regression. Ganguly et al. [26] propose a two-stage ensemble predictive model. Chaves et al. [25] design a failure prediction method using a Bayesian Network. Mahdisoltani et al. [23] explore a range of different machine learning techniques (Classification and Regression Trees (CART [49]), Random Forests (RF [50]), SVM, Neural Networks (NN [51]) and LR) and show that sector errors can be predicted ahead of time with high accuracy using RF. Carlos et .al [22] investigate three different machine learning models (Decision Trees, NN, and LR). Xiao et al. [21] introduce a novel disk failure prediction model using Online Random Forests (ORFs). Aussel et al. [24] test several learning methods, SVM, RF and Gradient-Boosted Tree (GBT) and find that RF provides the best performance. Anantharaman et al. [20] experiment with two different types of ML approaches: RF and Long Short-Term Memory (LSTM) recurrent neural networks. Yong et al. [19] develop a cost-sensitive ranking-based ML model that can learn the characteristics of faulty disks from the past and rank the disks based on their error-proneness using the FastTree algorithm [52]. Yi et al. [18] predict the failure order with a LambdaMART [53] model. Xie et al. [17] propose an explanation approach designed for disk failure prediction.

Deep Neural Network based Methods (DNN). DNN method is an artificial neural network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a nonlinear relationship. Xu et al. [34] introduce a method based on Recurrent Neural Networks (RNN) to assess the health statuses of disks. Pang [33] implement a Combined Bayesian Network (CBN) model to estimate the health degree of disks. Fernando et al. [32] present a remaining useful life estimation approach for disks by leveraging the capabilities of LSTM networks. Basak et al. [31] propose a data-driven framework based on LSTM for detection of whether a disk is going to fail in the next 7 days. Sun et al. [30] propose a temporal Convolutional Neural Network (CNN)-based model to predict failures.

One-Class Classification based Methods (OCC). OCC involves fitting a model on the normal data and predicting whether new data is normal or not. The imbalanced ratio of positive (failure) disk samples and negative (healthy) disk samples poses a significant challenge for efficient learning. Wang et al. [38, 39] present an anomaly detection method, which distinguishes anomalous behaviors from healthy events in each disk. Lucas et al. [37] propose a method for fault detection on hard disks that uses a Gaussian Mixture to model the behavior of only healthy hard disks. They also propose another method named GMFD [36], which applies a semi-parametric model (GMM) to build a statistical model using healthy disks. Francisco et al. [35] evaluate nine one-class classifiers for fault detection in hard disks.

Transfer Learning based Methods (TL). TL is an ML technique where a model trained on one task is re-purposed on a second related task. Transfer learning methods have been proposed to solve the minority disk failure problem. Botezatu et al. [43] apply a transfer learning approach to use a prediction model trained on a specific disk model for a new disk model of the same manufacturer. Francisco et al. [42] evaluate several transfer learning strategies on the task of failure prediction on hard disk drives. Their experiments state that transfer learning methods can provide performance gains in hard disk failure prediction models. Xie et al. [41] employ a simple but effective transfer learning method for disk failure detection which determines the best source from its anterior disk models and builds a transfer learning model with the help of the source domain. Zhang et al. [40,44] integrate both transfer learning and active learning techniques to detect the disk failure called ATAD and show its effective in cross-dataset time series anomaly detection. Zhang et al. [1] propose a minority disk failure prediction model named $TLDFP$ based on a transfer learning approach and use Kullback Leibler Divergence (KLD) as an effective indicator for source domain selection for large heterogeneous datasets.

Shortcomings of existing work. The above mentioned approaches (SML, DNN and OCC) deliver good detective performance only when both training and testing data are drawn from the same distribution [54]. A recent study [22] on heterogeneous disk failure prediction has pointed out that the predictive results are not good enough for adoption in prac-



(a) DAD approach [12]

(b) SML approach [17]

(c) DNN approach [31]

(d) OCC approach [35]

Figure 1: Figure 1(a), 1(b), 1(c) and 1(d) respectively show the overall F-Measure (detailed in Section 5.1.3) of the four state-of-the-art methods using datasets of disk models from two data centers. Although the SML, DNN and OCC achieved better performance than the DAD method in the case of datasets consisting of disks from the same disk models, the DAD approach delivers strong applicability and adaptability.

tice. Therefore, most of the experiments for these methods only consider datasets consisting of disks from the same manufacturer (and often of the same disk models) and thereby have limited applicability. Except for the method DAD, the methods (SML, DNN and OCC) learn the S.M.A.R.T data distribution of the specific disk model but not the unified detection measure, as a result, these approaches have bad predictive performance when dealing with the disk models that have not yet appeared in previously trained models (i.e., poor adaptability). Furthermore, all the methods (especially for DNN and OCC) require a large number of training samples to build robust models, which is difficult to be satisfied for minority disks in data centers. Training models on the minority disks would dramatically increase the risk of overfitting, and the resulting poor generalization will decrease the performance of predictive models [1]. Although TL approaches can handle this situation well, an important premise of this approach is that there is one or more appropriate source majority disk models for knowledge transfer, but we find this to be a tough assumption in practice (see Section 3.1.2). Moreover, most of these approaches increase the training cost to process the imbalanced datasets and could result in discarding a large amount of data or model overfitting (mentioned in Section 1).

3 Preliminary Study and Motivation

In this section, we investigate the performance of existing approaches in three aspects (applicability, adaptability and minority disk failure detection) and describe our motivation for enabling high-dimensional disk state embedding for heterogeneous disk failure detection.

3.1 Preliminary Study

3.1.1 Applicability and Adaptability

We analyzed the overall performance of existing proposed approaches using different disk models from both the publicly available S.M.A.R.T dataset Backblaze[1] and the data center of Tencent (one of the largest social network companies in the world). Figure 1(a), 1(b), 1(c) and 1(d) respectively show the overall detective performance of the state-of-the-art methods (DAD [12], SML [17], DNN [31] and OC [35]) using different datasets of disk models. For convenience, we use `Data Center-Disk Manufacturers-Disk Model` to denote the dataset we use. For example, the disk model C from Seagate in data center BackBlaze can be referred to as `B-STX-C` and the disk model A from WDC in data center Tencent can be referred to as `F-WDC-A`.

In each figure, the vertical and horizontal axes refer to the disk models of training and testing dataset respectively. We use F-Measure (detailed in Section 5.1.3) to evaluate the performance of failure detection models. The higher the value is (the darker in the heatmap), the better the performance is. In the last row of each figure, we use the hybrid datasets (the first 6 disk models contain F-STX-A, B-STX-B, F-WDC-A, B-WDC-B, F-HIT-A and B-HIT-B) for training and then detect on different disk models. Note that the last three disk models F-STX-C, F-WDC-C, B-HIT-C are not included in the hybrid disk models for training. We summarize the findings from these four figures. **(1)** Detection approaches (SML, DNN and OCC) built on a specific disk model only has good results test on the same disk model (i.e., the detective model built on F-STX-A and detect on F-STX-A, as illustrated by the darker diagonal lines). Moreover, DNN-based approach shows the best performance in this case. **(2)** The SML, DNN and OCC approaches built on hybrid disk model datasets decrease the detective performance compared to a model built on the same disk model (we call the model has poor applicability in this case). Moreover, the performance of these approaches further deteriorates when detecting the disk models that have not appeared in the training datasets. **(3)** Although the overall detective results of the DAD method is not good enough to be deployed in practice, it is not sensitive to the different disk models. In other words, the performance of cross-model disk failure detection is very close. Besides, the DAD approach built on hybrid disks increases the performance compared to the model built on a single disk model (good applicability) and shows high adaptability to disk models that have not appeared in training. Note that we also evaluated many other studies on these methods and got similar results and we don't discuss all the results here due to space limitations.

3.1.2 Minority Disk Failure Detection

In order to investigate the detection for minority disks (the number of disks less than 1,500 [1]), for TL approach, we use the majority disk model which has the smallest Kullback Leibler Divergence (KLD) values with the detecting minority

[1] https://www.backblaze.com/b2/hard-drive-test-data.html

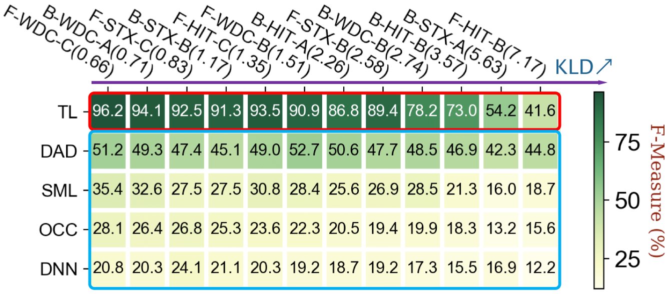


Figure 2: Minority disk failure detection using different approaches. The TL approach achieves the best performance but it depends on the small enough KLD value of the majority disk which is chosen for training.

disk model from the same manufacturer to train the detective models. The detecting minority disk models are listed on the x-axis ordered by the calculated smallest KLD values. For other approaches, they only trained the detective model based on minority disk datasets. Note that KLD is a metric measuring the divergence degree of one probability distribution from another expected probability distribution [55]. It indicates the disparities between two S.M.A.R.T datasets distributions. A zero KLD value means that the distributions of these two disk datasets are the same, while the KLD value increases as the differences between two data distributions widen. In general, the larger a KLD value is, the greater differences between two disk data distributions will be and the more difficult the knowledge transfer between two distributions will be in the transfer learning approach [1]. Figure 2 shows the results. The TL approach delivers the best detective performance on minority disk failure detection especially with the training datasets having smaller KLD values while other candidates are difficult to handle this situation.

Table 2: Distribution characteristics of the smallest KLD value for minority disk model in two data centers

Data Center	KLD(0∼1)	KLD(1∼2)	KLD(2∼3)	KLD(>3)
Tencent	**35%**	25%	23%	17%
Backblaze	**32%**	18%	31%	19%

In order to take an in-depth look at the feasibility of this method in practical large storage systems, we studied the disk quantities by the different KLD values in two data centers. As shown in Table 2, in Tencent data center, only 35% of all minority disk models could find a majority disk model with small KLD value (range from 0 to 1) for training. In other words, most minority disk models only find the majority disk models with a KLD value greater than 1 which might result in poor detection performance. A similar observation has been made in the data center Backblaze. Therefore, even in such large data centers with millions of disks, it is still difficult to find the most suitable majority disk model with small enough KLD value to use TL for minority disk failure detection.

3.2 Motivation

The preliminary study results above show that The DAD approach has better applicability and adaptability than other approaches (SML, DNN, OCC), and DNN approach gives the best detective performance. Before introducing our motivation, we first to answer the following three problems: **(1)** Why the DAD approaches have good applicability and high adaptability while DNN does not? The DAD method learns the distance (similarity) between the normal and abnormal disk samples in a certain space which has a commonality and not sensitive to the disk models. However, the DNN approach learns the distribution of S.M.A.R.T data which varies with disk models, the performance of this approach would decrease when the distribution changes. **(2)** Why the overall detective performance of the DAD method is not as good as other approaches. We think the reason is it performs distance-based transformation and computation in low-dimensional space but does not learn from the dynamically changed long-term behavior in high-dimension. **(3)** Why the DNN approach achieves the best performance among other candidates? DNN is good at mapping the raw low-dimensional S.M.A.R.T attributes to high-dimensional target spaces through complex transformations that perform good expression and fitting ability and thus achieves better performance.

Considering the above advantages of the DAD and DNN approaches, we are motivated to apply the distance-based anomaly detection approach and deep learning to build a general disk failure detection system for heterogeneous disks. In order to learn a unified measure of distance in high-dimension space using deep learning and then easily use the distance-based anomaly detection approach for comparison for two input data, we applied the siamese networks [56] (commonly used in the image recognition technology) which we will describe in detail in the next section.

4 Proposed System *HDDse*

In this section, we first provide an overview of our proposed system *HHDse* in Section 4.1. Then, we introduce the LSTM-based siamese network for disk failure detection in Section 4.2. Finally, we describe the sample pool and decision maker of *HDDse* respectively in Section 4.3 and Section 4.4.

4.1 System Overview

Figure 3 provides an overview of our proposed novel system *HDDse* which combines a sample pool, an LSTM-based siamese network for disk failure detection, and a decision maker. **Sample pool** (see Section 4.3 for details) is used to store the S.M.A.R.T instances collected daily for each disk. These instances are combined to form training and detecting samples for our approach. Note that the sample pool has both the disk S.M.A.R.T instances with ground truth (true labels with the disk states) for training and the unlabeled instances for detecting. **LSTM-based siamese network for disk failure detection** (see Section 4.2 for details) is the core part in *HDDse*, the input of the network is a pair of samples from the sample pool and the output is a binary classification result that indicates the similarity of these two samples. In online

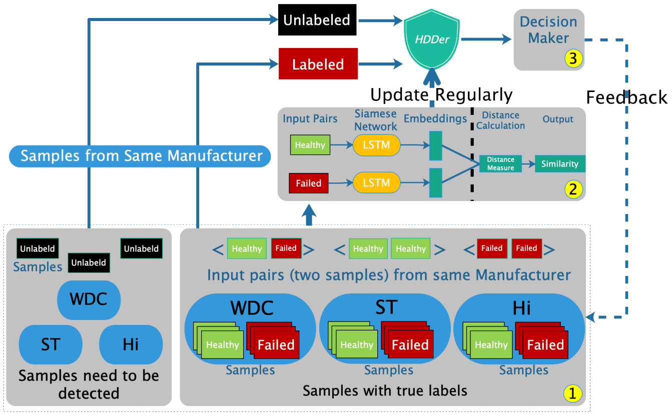


Figure 3: The overview of the proposed *HDDse*. It consists of a sample pool, an LSTM-based siamese network and a Decision Maker marked as 1, 2 and 3 respectively.

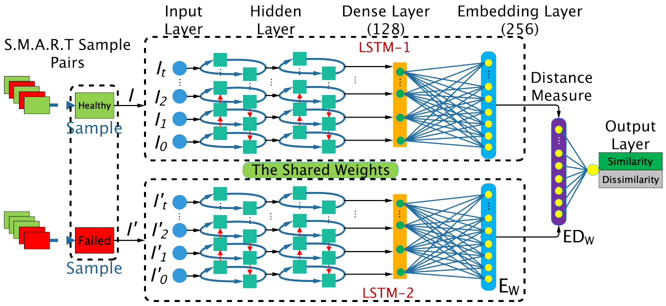


Figure 4: The structure of our proposed LSTM-based Siamese Network for Disk Failure Detection. The input is a pair of detecting samples and output is similarity.

detection, since there are many results of the target detecting samples at a continuous moment compared with other labeled samples, each output detective result indicates for a particular sample at different moments rather than the whole disk health state. Therefore, **Decision Maker** (see Section 4.4 for details) is a module to map these sub-results of samples to the final whole disk healthy state.

4.2 LSTM-based Siamese Network

Many previously proposed machine learning models, such as random forests (RFs), decision trees, SVM and DNN rely on the phenomenon that some key attributes are distinct from others when the disk is going to fail. Therefore, these approaches take a single snapshot of S.M.A.R.T attributes as training data for detection, without considering the sequential dependency between different statuses of a hard disk over time (only rely on features extracted from one day) because they are unable to make use of the time series data (except for converting them to sequential features manually). However, many researches have shown that the S.M.A.R.T of the disk changes dynamically with a certain trend [2, 19, 21, 27, 57], thus the current state of the disk may depend on a long-term historical trend. In contrast to all the aforementioned methods, we apply the LSTM to the parsing of sequential S.M.A.R.T information. LSTMs have been successfully applied to a variety of applications, including text sentiment classification [58] and multi-language text classification [59].

Traditional approaches to solving a classification problem, such as SVM, random forests (RFs) or even DNN, generally require that all the categories be known in advance (some newly disk models entered the data center without enough history information to classify the categories) for training. Moreover, those approaches are not suitable for applications where the number of samples per category is small (minority disk models). Siamese networks are dual-branch networks with tied shared weights, i.e., they consist of the same network copied. This method is proposed for training a similarity metric from data, which can be used for classification tasks (eg., face recognition) in which the categories need to be classified have not appeared during the training process, and can also handle the situation of training samples for a single category which is very small [60]. The symmetry of siamese networks is important and reasonable for learning a unified measure of distance, for example, the distance from disk state S_1 to state S_2 should be equal to the distance from S_2 to S_1.

Therefore, we propose an LSTM-based siamese network for disk failure detection. Figure 4 shows the structure of the network. The training sample pairs we feed to the network are randomly selected from the sample pool which contains two S.M.A.R.T samples (see Section 4.3 for details) from the same disk manufacturer denoted as $< S, S' >$. We design two LSTM networks to receive these two S.M.A.R.T samples respectively which need to be compared for their similarity. Note that these two LSTM networks (BiLSTM [61] refers to bi-direction LSTM layers) shared their weights in order to map the inputs to the same target high-dimension space for comparison. Each LSTM consists of an input layer U, four hidden layers H, one dense (fully connected) layer D with 128-dimensional units and an embedding layer E. In contrast to traditional neural networks, the LSTM operates over sequences of input vectors. This structure is able to capture the historical context of disk healthy statuses and makes LSTMs suitable for tasks related to sequential detection. Note that the dense layer is followed by an embedding layer (a fully connected layer) with 256-dimensional units, and then one more layer computing the distance metric (Euclidean distances [2]) between each siamese twin network. We employ a *Sigmoid* function in the final layer to output a normalized value on the learned high-dimensional feature space and scores the result between the feature vectors of the input sample pair. Note that our proposal is the first to propose an LSTM-based Siamese network based on the peculiar features of disk failures and has shown promising prediction outcomes.

Learning process. Let $E_{\omega}(S)$ and $E_{\omega}(S')$ be the projections of the input pair S and S' in the embedding space computed by the embedding layer (high-dimension space) network function E_{ω}. The output layer is a single unit computed by the induced distance metric between each siamese twin. The detective

[2] It is interesting to explore other sophisticated distance metrics to achieve better results.

vector is given by:

$$ED_\omega(S,S') = f_{sig}(\sum_{i=0}^{N-1} \beta_i |E^i_\omega(S) - E^i_\omega(S')|))$$

where f_{sig} is a *Sigmoid* activation function. β_i are additional parameters that are learned by the model during training, weighting the importance of the component-wise distance. Supposing that N represents the total number of S.M.A.R.T sample pairs over a dataset $D = < S_i, S'_i, Y_i >$, where i indexes the ith training pair and $Y(S^i, S'^i)$ is the corresponding label. We assume $Y(S_i, S'_i) = 1$ whenever S^i and S'_i are the same disk state label and $Y(S_i, S'_i) = 0$ otherwise. The total loss function is given by:

$$L_\omega(D) = \lambda||\omega||_2 + \frac{1}{2N} \sum_{i=0}^{N-1} \ell^i_\omega(S_i, S'_i, Y_i)$$

We use a squared L2-norm regularization (also called ridge regression) in this loss function to improve the ability of model generalization. The ω are the weights of the neural network and λ is the weight decay (this prevents the weights from growing too large and can be seen as gradient descent on a quadratic regularization term to prevent from the model overfitting) set as 0.001 to train our model. The instance loss function ℓ^i_ω comprises of terms including the similar (Y = 1) case (L_s), and the dissimilar (Y = 0) case (L_d):

$$\ell^i_\omega = Y_i \ell_s(S_i, S'_i) + (1 - Y_i)\ell_d(S_i, S'_i)$$

The loss functions for the similar and dissimilar cases are given by:

$$\ell_s(S,S') = (ED_\omega)^2$$

$$\ell_d(S,S') = \begin{cases} (m - ED_\omega)^2, & m < ED_\omega, \\ 0, & otherwise. \end{cases}$$

m is a margin which defines how far away the dissimilarities should be. These settings were chosen during cross validation, grid searching over possible margin settings. Suitable margin helps us distinguish the two input S.M.A.R.T samples better. Therefore, the instance loss function can also be given by:

$$\ell^i_\omega = Y_i(ED_\omega)^2 + (1 - Y_i)\{max(m - ED_\omega, 0)\}^2$$

It is interesting to investigate the proposed loss function. When the two samples are similar/dissimilar (Y = 1/ Y=0), if the ED_ω is wrongly calculated large/small by our model, the value of this loss function will become larger. Our goal is to minimize it.

The parameters of our model are optimized using the Adam optimizer [62] with a decreased decaying learning rate. The training process was run for 150,000 epochs with learning rate (lr) starting at 0.1 and decrease it by a factor of 2 every 50 training epochs. We use the dropout technique [63] (a dropout of 0.5) used on the recurrent units and between layers to prevent overfitting. For the hyper parameter (margin m, weight decay λ, learning rate, the unit number of fully-connected layers) optimization, we use the grid searching to perform hyper parameter selection.

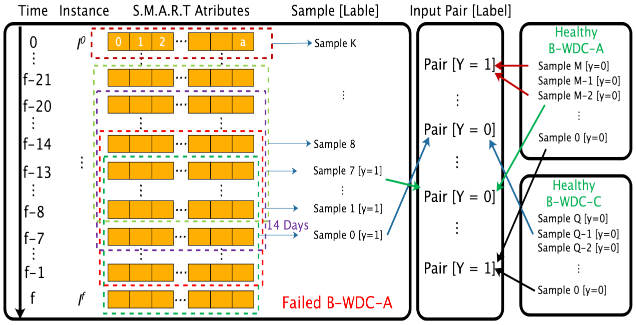


Figure 5: The relationship between S.M.A.R.T instances, samples and the input pairs in our approach. Each sample consists of 14 continuous instances and the continuous samples in a period of seven days before actual failures are labeled as failed.

4.3 Sample Pool for Imbalanced Datasets

Our sample pool consists of the collected S.M.A.R.T instances with their corresponding confirmed labels and some instances that need to be detected. We collect all the instances of each disk in the data center every day. All the time data have been discretely sampled at an interval of one day. If the data collection starts at day T_0 and a disk can fail in any day T_f after that, a disk can have data for all such time indices t where t varies from T_0 to T_f. We can formulate this instance as a multivariate sequential vector $I^t = \{I^t_0, I^t_1, I^t_2, ..., I^t_a\}$ that contain length-a attributes of S.M.A.R.T data at time t for each disk. More specific information about the attributes we use in our evaluation is given in Table 3. Besides, each input of the LSTM consists of a fixed length (we set 14 days in our system to learn the long-term disk state behavior) continuous instances (referred to one sample) with its corresponding label. The first day recorded in a disk failed sample is not the day when the disk fails, but the latest day when the collected attribute stops changing [2, 21, 23]. We use change-point detection to decide how many days samples of failed disks preserve for training. Change-point detection can find an abrupt change during a period. We observe that most attributes have a significant change around 7 days before failures. Therefore, for a failed disk, continuous samples in a period of 7 days (our method has the ability to predict the failure 1-7 days in advance) before actual failure (T_f) are labeled as failed ($y = 1$). For healthy disks, we label the total continuous samples as healthy ($y = 0$). Note that Y = 1 (Y = 0) means the label of the generated input pairs which is a combination of two same (different) labeled samples. The relationship between S.M.A.R.T instances, samples and the input training pairs is shown in the Figure 5. RAID arrays will not affect the operations in the sample pool and the decision maker in Section 4.4.

As mentioned in Section 4.2, the inputs of our proposed network are two S.M.A.R.T samples $< S, S' >$ in pairs. Therefore, we can freely combine and label a pair of samples from the same disk manufacturer. There are two benefits for generating this form of training datasets compared to existing approaches which treat each S.M.A.R.T sample as a snapshot in the training process.

Better with imbalanced datasets. It reduces the degree of data imbalance by the simplest free combination. We use the imbalance degree (*IDe*) [64] to indicate the dataset imbalance which is defined as the ratio between the majority and minority samples (the larger value the *IDe* is when *IDe* is larger than 1, the more imbalanced the dataset is). For an imbalanced dataset containing a minority class sample with size A and the IDe is α, the majority class sample size is αA. The number of pairs with labels $Y = 1$ ($Y = 0$) n_1 (n_0) after combining the input samples can be expressed as: $\begin{cases} n_1 = C_A^2 + C_{\alpha A}^2, \\ n_0 = C_A^1 \times C_{\alpha A}^1. \end{cases}$ where the C_A^2 ($C_{\alpha A}^2$) is the number of "similar" sample pairs generated by the minority class samples (majority class samples). The new imbalance degree IDe' of the input pairs is given by: $IDe' = \frac{n_1}{n_0} = \frac{\alpha}{2} - \frac{1+\alpha-A}{2A\alpha}$ since $A, \alpha > 1$, the value of IDe' will be around $\frac{\alpha}{2}$, which effectively alleviates the original data imbalance by a factor of two. Note that we directly arrange all labeled samples to form the input training pairs without losing large amounts of information (instances) in the sample pool compared to the Under-sampling method which is commonly used in recent researches. Although our method can alleviate imbalanced datasets, it is difficult to eliminate. Fortunately, extensive experimental results in Section 5 demonstrate that the siamese network is insensitive to imbalanced datasets, which is the same as many existing research results [65, 66].

Better with minority disk models. It forms large training samples even with the minority disk models. The number of training pairs with the minority disk models in existing methods is $P = A(1+\alpha)$. However, in our method the number of training pairs is: $\frac{P!}{2!(P-2)!} = \frac{P(P-1)}{2}$ which is extremely large compared to existing methods. Therefore, our model can increases the number of samples greatly and make better use of deep learning algorithms to detect the failure and avoid model overfitting (the experimental results are discussed in Section 5.2.2).

4.4 Decision Maker in *HDDse*

In this section, we aim to seek answers to the following two problems: **(1) Which** training samples need to be arranged to form the input pairs with the detecting sample and how to make a decision (same state or not) for these pairs in the online detection process. **(2) How** to make a disk healthy state decision for a target disk based on the results of these detecting samples at continuous moments?

When a sample of a target detecting disk needs to be detected, we let all the labeled training samples from the same disk manufacturer to be arranged to form the input pairs. For accelerating the process of making a decision for each sample, we follow four detecting steps:

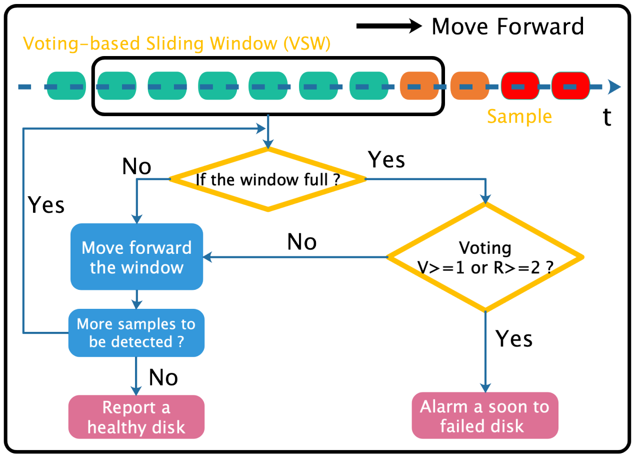


Figure 6: The flowchart of the voting-based sliding window. Moving forward until the disk health statue is reported.

Step 1. We arrange all the samples labeled as failed ($y = 1$) from the same disk model (all majority and minority disks) to form the input pairs with the target detecting sample respectively. If any of these pairs are detected as similar ($Y = 1$), we will mark this detecting sample as failed ($y = 1$), otherwise, proceed to the next step.

Step 2. Considering the sparsity of the state of healthy disk samples, we first randomly select 10% healthy disks of the same disk model and arrange the samples ($y = 0$) collected randomly in one-month intervals to form the input pairs with the detecting sample. If all these pairs are detected as similar ($Y = 1$), we will mark this sample as healthy ($y = 0$). Otherwise, follow the third step.

Step 3. We compare with the samples labeled as failed ($y = 1$) from the different disk models but the same manufacturer with the detecting sample. If more than half of the results are detected as similar ($Y = 1$), we will mark this sample as failed ($y = 1$).

Step 4. Similar to step 2, the only difference is we arrange the different disk models from the same manufacturer. Note that when the disk model of the detecting sample did not exist in the sample pool, we will start from step 3.

Each target detecting disk consists of several results of detecting samples at continuous moments, each output result indicates the detection for a particular sample at one moment rather than the whole disk health state in a period. Therefore, to improve the robustness of the detection method against noise, we propose a voting-based sliding window (VSW) method to make a disk healthy state decision for the final disk state. Figure 6 shows the architecture of VSW. According to the results of the detecting samples stored in the Decision Maker, we define the VSW method for failure detection in the following manner: Define a length-W time sliding window and move it forward everyday. A failure alarm will be reported, if the window consists of V (R) consecutive results from step 1 (step 3), otherwise, there is no failure alarm. These parameters (W, V, R) will be optimized as hyper parameters of the model to determine its optimal value and were chosen during cross validation, grid searching over possible settings. The configuration of our real-world large scale storage system is $W = 7, V = 1, R = 2$. Note that some other decision solutions can be explored to improve the robustness of our method further (eg., a weighted k-nearest neighbors approach).

5 Experimental Evaluation

In this section, we evaluate the detective performance of *HDDse*. We first describe the methodology, followed by the experimental results of effectiveness and efficiency and compare *HDDse* against the state-of-the-art approaches with respect to the evaluation metrics.

5.1 Methodology

We describe the characteristics of two real world S.M.A.R.T datasets and the attributes selection in our experiments. Then we introduce the experiment setup and some evaluation metrics used in our experiments.

5.1.1 Datasets and Attribute Selections.

Datasets. We use S.M.A.R.T datasets from two real world data centers for evaluation. One is the publicly available data set from "Backblaze"[3], which spans a period of 58 months consisting of 146,203 healthy disks and 8,256 failed disks. The second proprietary dataset has been collected by Tencent and spans 29 months consisting of 70,192 healthy disks and 2,971 failed disks. All disks in these datasets were labeled to be either failed or healthy. Note that the data from these two data centers are extremely imbalanced. We tried to impute the missing S.M.A.R.T values or disks by replacing them with the median value.

Attribute Selections. Each S.M.A.R.T instance contains many meaningful attributes. We first select all the common attributes of the disk manufacturers. However, we find some attributes are irrelevant to disk failure events because they are immutable or have not experienced noticeable abnormal changes. Therefore, we use correlation coefficients and select nine attributes that correlate most with disk failure. The selected attributes are listed in Table 3. Each SMART attribute entry consists of many elements. In our paper, we focus on the three elements $(ID, Normalized\,value, Raw\,value)$ in our collected datasets. Since different attributes have different output ranges, (which might lead to different impacts on the detection model), we normalize the range of all S.M.A.R.T attributes using min-max normalization, a common preprocessing technology in machine learning: $x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$ where x is the original value of a S.M.A.R.T attribute, x_{max} and x_{min} are the maximum and minimum value of the attribute in the training data set, respectively. Note that we tried other normalization methods (e.g., z-score), but achieved the best results using min-max normalization. The hyper parameters and attribute selections were done over the entire datasets.

Table 3: The S.M.A.R.T Attributes for Our Evaluations

#ID	S.M.A.R.T Attribute Name	Attribute type
001	Raw Read Error Rate	Normalized
004	Start/Stop Count	Raw
005	Reallocated Sectors Count	Raw
012	Power Cycle Count	Raw
187	Reported Uncorrectable Errors	Normalized
193	Load Cycle Count	Normalized
196	Reallocation Event Count	Raw
197	Current Pending Sector Count	Raw
198	Offline Uncorrectable Sector Count	Raw

5.1.2 Experiment Setup.

To simulate the detecting process of disk failure in the real world, we use the following method to build the experimental datasets. All disks are randomly divided into a training set and a testing set at a ratio of 7 to 3, as in most researches [1,35,67, 68] which guarantees that the failed disks in the training set and the testing set are completely independent. Note that the deterioration of a failed disk is a gradual process from healthy to failure, and not all samples of failed disks need to be used in the training set; otherwise, those healthy samples of failed disks which are far from the actual failure would disturb the training of the detection model. Therefore, only the last seven continuous samples (we detailed it in Section 4.3) before the moment of failure of the training disks can be regarded as failed samples and need to be added to the training dataset. Furthermore, we obtain all results via cross-validation [69] to decrease the variability of the detections (analogous to many methods [1,41,70]). For the configurations and parameters of our system *HDDse*, the maximum number of training epochs is set to 1000; the learning rate is initially set to 0.1, and we decrease it by a factor of 2 every 50 training epochs; the coefficient of weight decay λ is set to 10^{-8}. We train and evaluate our method on a Linux server with 12-core 4.0GHz CPU, 64GB RAM and 200GB HDD.

5.1.3 Evaluation Metrics.

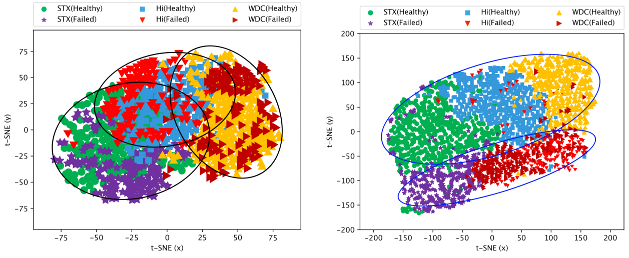
We use the following five metrics to report the detective performance in our experiments which are commonly used for evaluating the capability of disk failure detection approaches.

TPR. True Positive Rate (also called recall) is the proportion of failed disks that are correctly predicted. The higher the TPR is, the better the model is.

FPR. False Positive Rate (also called false alarm rate) is the proportion of healthy disks that are falsely predicted as failed. The lower the FPR is, the better the model is.

AUC. We use the AUC (Area under the receiver operating characteristic curve) value under the ROC curve (receiver operating characteristic) to evaluate the binary classification performance of our detection model in imbalanced datasets. ROC is a curve plotting the TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis. Therefore, the larger the AUC value, the higher the TPR and the lower the FPR. AUC is the area under this curve. A higher AUC means the model is better at distinguishing failed and healthy disks.

[3] https://www.backblaze.com/b2/hard-drive-test-data.html



(a) Before embedding. (b) After embedding.

Figure 7: The t-SNE of the S.M.A.R.T data before and after embedding using our approach. After the disk state embedding, the healthy and failed disks are nearly clustered together and easily separated.

F-Measure. F-Measure is a balance between the two metrics TPR and Prediction Precision (PP). PP is the proportion of detected failed disks that are correctly detected. The higher the F-Measure is, the better the model is.

C-MTTDL. We use Cost-based MTTDL (Mean Time To Data Loss) to evaluate the reliability and availability of the storage system. MTTDL [71] was proposed to approximate the mean time to data loss with failure detection model which is given by: $MTTDL \approx \frac{MTTF}{1-\frac{k\mu}{\mu+\gamma}}$ where $MTTF$ is the Mean Time To Failure of a disk. k is the TPR and γ is the inverse of how far in advance the model can detect the impending failures. μ is the inverse value of Mean Time To Repair (MTTR, replace a new disk or transfer the data of the failing disk to a new one). However, this metric only demonstrates the relationship between the MTTDL and the TPR (correctly detected) but neglects the cost of misclassification by the approach. Too many misclassifications will result in unavailability of the storage system. Therefore, we propose an end-to-end economic analysis metric called the Cost-based MTTDL ($C-MMTDL$) which considers not only the reliability but also the misclassifications cost (considering the false positive rate and cost of replacing disks). We will give the detailed definition of $C-MTTDL$ in Section 5.2.5. Although we propose the $C-MTTDL$, it is interesting to explore other end-to-end evaluation metrics (benefit in customer-perceived latency or throughput from accurate prediction).

5.2 Effectiveness Comparison

In this section, we first analyze the disk state embeddings learned by our LSTM-based Siamese Network. Then we show the results of *HDDse* compared to several state-of-the-art approaches in aspects of the ability of the minority disk detection, model applicability and adaptability respectively.

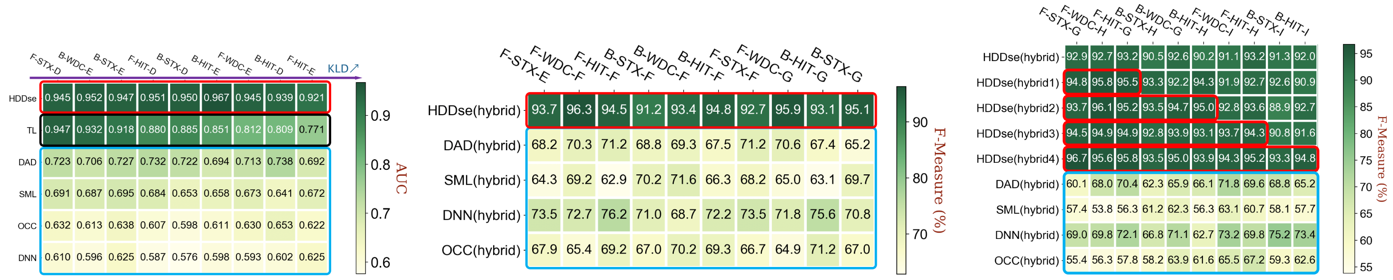
5.2.1 Analysis of Disk State Embeddings

To visualize the high-dimension embedded disk statues of the embedding layers in our approach (See Figure 4), we use the t-Distributed Stochastic Neighbor Embedding (t-SNE) [72] technique which can project high-dimensional embedding spaces into 2D spaces for visualization while striving to keep data clustered together in the high dimensional space clustered together in the low-dimensional space as well.

For comparison, Figure 7 shows the t-SNE of the S.M.A.R.T data attributes in low and high dimensionality respectively (before and after embedding using *HDDse*) based on the collected data from three disk manufacturers. In our case, the low dimensionality is the original S.M.A.R.T attributes most of the recently proposed methods (DAD, SML, DNN, OCC and TL) leveraged (the specific attributes are listed in Table 3). Note that the x and y axes of a particular point have no meaning on their own and the t-SNE only attempts to preserve clusters in higher dimensional space. The data points have been colored and shaped based on disk statues and their manufacturers. It is observed the data points of these three disk manufacturers cluster relatively closely and the relationship between the healthy and failed disks is hard to distinguish with a unified method shown in Figure 7(a). This highlights the challenges in failure detection of disks based on S.M.A.R.T attributes in heterogeneous populations. This result is consistent with the findings of the research [70] and results in most of the related works only considered population consisting of the same disk models in their experiments. As can be seen from Figure 7(b), the healthy and failed disks are easily separated (the healthy state is above the failed state for all the disks from different manufacturers) after the embedding using our proposed approach. This experiment demonstrates that our *HDDse* can generate a unified and efficient high-dimensional disk state embeddings for generic disk failure detection of heterogeneous disks.

5.2.2 *HHDse* only Trained on Minority Disk Datasets.

As mentioned in Section 4.3, the input training pairs generated by our method is extremely large compared to the original samples of existing approaches. It is interesting to investigate the detective performance of *HHDse* when only trained on minority disk datasets. We compare our approach with the other five state-of-the-art approaches: DAD [12], SML [17], OCC [35], DNN [31] and TL [1]. For a fair comparison, the TL approach uses the majority disk model which has the smallest KLD value with the detecting minority disk model from the same manufacturer to train the detective models. Minority disk models from different data centers and manufacturers are listed in the x-axis ordered by these calculated the smallest KLD values. For other approaches, they only trained the detective model based on minority disk datasets. As can be seen from Figure 8(a), none of the four approaches (DAD, SML, OCC and DNN) can deliver a high AUC value. The poor detective performance is due to overfitting caused by using small homogeneous datasets [1]. In particular, DNN achieved the worst results because the neural network required a huge number of samples to fit a large number of weights. Besides, the results of the TL method imply it largely depends on whether you can find the smallest KLD value majority disk training dataset for modeling as discussed in Section 3.1.2. It is worthy to note that our *HDDse* achieves the best performance in all cases. The reason is two-fold. On the one hand, the training pairs generated by our method are extremely large which makes our model not easy to get overfitting. On the other



(a) (b) (c)

Figure 8: (a) Performance comparison for minority disk failure detection of different approaches. *HDDse* delivers the highest AUC in most cases. (b) Performance comparison for disk failure detection using hybrid disk datasets. *HDDse* achieves the best detection results in all cases which shows good applicability. (c) Performance comparison for disk failure detection using different hybrid disk datasets. *HDDse* achieves the best detection results even the disk models have not appeared for training which shows good adaptability.

hand, the disk state embeddings in high-dimension learned by our designed LSTM-based Siamese Network is effective for model classification.

5.2.3 The Applicability of *HHDse*

In order to investigate the model applicability of our *HDDse*, we use the imbalanced datasets of ten disk models from three disk manufacturers in two data centers to conduct this experiment. We compare our approach with the other four approaches: DAD, SML, OCC and DNN. Considering the generality, those ten detecting disk models consist of three from the data center Tencent (F-STX-E, F-WDC-F and F-Hi-F), three from data center BackBlaze (B-STX-F, B-WDC-F, B-HIT-F) and four minority disk models from two data centers (F-STX-F, F-WDC-G, B-HIT-G, B-STX-G). We use all these hybrid disk datasets to train and detect these models respectively, and the detective results are shown in Figure 8(b). When dealing with the disks from different manufacturers, data centers and even minority disk models, *HDDse* achieves higher F-Measure values compared to other candidates. The main reason is that our model maps low-dimensional attributes (disk status) to a general high-dimensional space that is not sensitive to the differences in disk models, which leads to the detection to be performed well using a unified distance calculation. This experiment demonstrates that *HDDse* has good applicability for disk failure detection in a heterogeneous environment.

5.2.4 The Adaptability of *HHDse*

We evaluate the adaptability of our *HHDse*, e.g., how an approach can adapt to a disk model that has not appeared in the training dataset. We use another ten disk models (note that they are different from the ten disk models showed in Figure 8(b)) from three disk manufacturers in two data centers to conduct this experiment. As can be seen from Figure 8(c). Data set “Hybrid” contains the same ten disk models for training described in Figure 8(b). “Hybrid1” contains another three disk models from data center Tencent (F-STX-G, F-WDC-H and F-Hi-G) compared to “Hybrid”, dataset “Hybrid2” add another three from data center BackBlaze (B-STX-H, B-WDC-G, B-Hi-H), “Hybrid3” has two more minority disk models from data center Tencent (F-WDC-I , F-HIT-H) than “Hybrid2” and “Hybrid4” includes all 10 disk models in the x-axis compared to the dataset “Hybrid”. For a fair comparison, we use the dataset “Hybrid” to train the DAD, SML, DNN, OCC approaches. The results are shown in Figure 8(c). Our method using different hybrid datasets all achieved better detective performance than other candidates. It is worthy to note that the datasets of the detecting disk models have not appeared for training delivers comparable performance as those disk models contained (framed by red lines) in training data sets in the first five rows of Figure 8(c). It indicates that our *HDDse* does not need to establish or maintain a new model and owns strong adaptability that can completely adapt to a new disk model regardless of disk manufacturers, disk models, data centers, and minority disks. Note that we also verified the HDDse’s performance on NVMe SSD and obtained promising results, which are not included due to the space limit.

5.2.5 Improvement of Storage System Reliability

As the above comprehensive analysis of detective results described, we achieved better AUC and F-Measure compared to other approaches. In this section, we will first give the definition of the economic analysis metric $C-MTTDL$ and then quantitatively evaluate the reliability and availability of the system based on different approaches. $C-MTTDL$ can be expressed as: $C-MTTDL = \frac{MTTDL}{Cost} \approx \frac{MTTF}{(1-\frac{k\mu}{\mu+\gamma})(C_aFP+C_bFN)}$ where MTTDL was defined in Section 5.1.3 and FP (FN) is the number of true healthy (failed) disks that are falsely predicted as failed (healthy). C_a and C_b are the corresponding cost of these misclassifications. These two factors can be set differently and obtain different results according to the model maintenance requirement (cost sensitivity) in the real world. C_a and C_b are set to 200 and 100 (dollars) respectively in our evaluation which were estimated in Tencent data center. $C-MTTDL$ denotes the MTTDL per dollar cost and the larger the better. The larger the $C-MMTDL$ is, the better is the reliability and availability the storage. We investigate the $C-MTTDL$ in based on the datasets from data center Tencent. We assume all models can detect the impending failures seven days in advance ($\gamma = 1/(7*24hours)$, the inverse value of $MTTR$ $\mu = 1/10hours$ and $MTTF$ = 1,390,000 hours. All these parameters come from data center Tencent and our experiments. We list the results in Table 4, which shows our approach improves the $C-MTTDL$ by about 2 orders of magnitudes than the other four candidates. In other words, in addition to ensuring the detective performance of our approach *HDDse*, we have greatly improved the reliability of the storage system at a lower cost.

Table 4: Improvement of C-MTTDL

Method	$k(TPR)$	**FP**	**FN**	**Cost**	**MTTDL (years)**	**C-MTTDL (hours/dollar)**
OCC	**62.6%**	8212	1062	1,748,600	397.6	1.94
DAD	**45.2%**	3422	1537	838,100	276.7	2.89
SML	**72.6%**	6159	783	1,310,100	504.10	3.37
DNN	**85.3%**	4791	419	1,000,100	814.13	7.13
HDDse	**95.8%**	103	140	34,600	1656.3	419.35

5.3 Efficiency Comparison

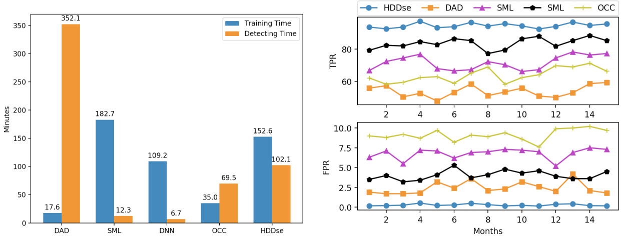
In this section, we evaluate the effect on training/detecting time and the practical long-term availability of our approach *HDDse* compared to the state-of-the-art approaches.

5.3.1 Training and Detecting Time

We train and evaluate all the approaches on a Linux server with 12-core 4.0GHz CPU, 64GB RAM and 200GB HDD. All the approaches record the total training (from start until convergence is achieved, i.e. little changes in the performance) and online detecting time (it is for the entire testing set). Figure 9(a) provides further time statistics for each approach. DAD has the lowest training time and the highest detecting due to its easy training method and large computation of calculating distances. Our approach *HDDse* takes the second-highest training time followed by the SML method (Random Forests) and the second-highest detecting time followed by the DAD method. This is attributed to our method generating large training pairs compared to the original datasets that need more time to converge. Moreover, it needs more time compared to other training samples to determine the disk statue in the detecting process (we use some sampling methods to accelerate the process of detecting mentioned in Section 4.4). Considering the training task in a data center and sometimes, the detective models are updated weekly or monthly (we perform once a week), so the time cost of *HDDse* is acceptable. Note that the time cost can be further shortened if GPU and many model compression and acceleration technologies [73] are used. We leave in our future work to explore other solutions for optimizing the efficiency of our approach.

5.3.2 Evaluating Practical Long-Term Availability

We last simulate practical long-term availability in data centers Tencent. As we mentioned in section 4.2, S.M.A.R.T data of the disk changes dynamically with a certain trend, thus the distribution of S.M.A.R.T attributes changes over time, resulting in unstable detective performance for many approaches. However, it is important to design a stable detection approach for large data centers without additional manual model tuning. To fairly evaluate the detective performance of different approaches in the long term, we employ the same accumulation strategy to update the model periodically e.g., once a week, using all the training data collected from the beginning. Figure 9(b) shows the TPRs and FPRs of the different detective approaches in the following 15 months. As can be seen, *HDDse* shows higher TPR and lower FPR compared to other state-of-the-art methods. It is worthy to note that *HDDse* exhibits stable detective performance than other candidates. We attribute this to the LSTM-based network which is well learned from the dynamically changed long-term behavior of disk statues. Similar to most of related works, we are focused on the predictive accuracy, because designing an accurate approach is the first critical step toward building a robust, highly reliable, and readily available operational storage system. With a high-accuracy failure prediction approach in place, we will have a high level of confidence in integrating it into the system, which is more of a mechanism rather than a policy. For instance, a direct application is to use the prediction results to perform data backups and replace disks that are about to fail to prevent data loss. Moreover, we can use the predictive results to analyze the mechanism of disk sector error and build a sector error predictive model to accelerate the scrubbing rate of disks to find the sector errors in advance and improve the reliability of the storage system.



(a) (b)

Figure 9: (a) Training and detecting time comparison for different approaches. (b) TPR and FPR of different approaches. *HDDse* delivers higher TPR and lower FPR in a stable manner which shows long-term availability.

6 Conclusion

Disk failures have become one prevailing reason for unexpected system unavailability. In this paper, we propose *HDDse*, an LSTM-based siamese network that can learn the dynamically changed long-term behavior of disk healthy statues and generate a unified and efficient high dimensional disk state embeddings from low dimensional S.M.A.R.T attributes for disk failure detection. We evaluate our approach using two real-world datasets to demonstrate that *HDDse* is effective and outperforms several state-of-the-art approaches. Specifically, *HDDse* has good detective adaptability to the disks which have not appeared in training and deliver good performance for the imbalance or minority disk datasets, thus improving storage system availability. Furthermore, the proposed approach improves the reliability of a data center and exhibits long-term availability.

Acknowledgments

We thank the anonymous reviewers and our shepherd Eno Thereska for their help in improving our paper. This work is supported by the Innovation Group Project of the National Natural Science Foundation of China No.61821003.

References

[1] Ji Zhang, Ke Zhou, Ping Huang, Xubin He, Zhili Xiao, Bin Cheng, Yongguang Ji, and Yinhu Wang. Transfer learning based failure prediction for minority disks in large data centers of heterogeneous disk systems. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP, pages 66:1–66:10. ACM, 2019.

[2] Xiaoyi Sun, Krishnendu Chakrabarty, Ruirui Huang, Yiquan Chen, Bing Zhao, Hai Cao, Yinhe Han, Xiaoyao Liang, and Li Jiang. System-level hardware failure prediction using deep learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 20:1–20:6. ACM, 2019.

[3] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. Cluster storage systems gotta have heart: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 345–358. USENIX Association, 2019.

[4] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, pages 15–26, 2012.

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, and Arild et. al Skjolsvold. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157. ACM.

[6] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116. ACM.

[7] Jayanta Basak and Randy H. Katz. Significance of disk failure prediction in datacenters, 2017.

[8] Drew D. Penney and Lizhong Chen. A survey of machine learning applied to computer architecture design, 2019.

[9] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. Raidshield: Characterizing, monitoring, and proactively protecting against disk failures. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 241–256. USENIX Association, 2015.

[10] Allen Bruce. Monitoring hard disks with smart. *Linux Journal*, 117, 2004.

[11] Richard Nass. Smart failure-prediction method now being endorsed for scsi disk drives. *Electronic Design*, 43, 1995.

[12] X. Gao, S. Zha, X. Li, B. Yan, X. Jing, J. Li, and J. Xu. Incremental prediction model of disk failures based on the density metric of edge samples. *IEEE Access*, 7:114285–114296, 2019.

[13] Yu Wang, Qiang Miao, and M. Pecht. Health monitoring of hard disk drive based on mahalanobis distance. In *2011 Prognostics and System Health Managment Confernece*, pages 1–8, 2011.

[14] Y. Wang, Q. Miao, E. W. M. Ma, K. Tsui, and M. G. Pecht. Online anomaly detection for hard disk drives based on mahalanobis distance. *IEEE Transactions on Reliability*, 62(1):136–145, 2013.

[15] Jing Shen, Jian Wan, Se-Jung Lim, and Lifeng Yu. Random-forest-based failure prediction for hard disk drives. *International Journal of Distributed Sensor Networks*, 14(11), 2018.

[16] S. Huang, S. Fu, Q. Zhang, and W. Shi. Characterizing disk failures with quantified disk degradation signatures: An early experience. In *2015 IEEE International Symposium on Workload Characterization*, pages 150–159, 2015.

[17] Y. Xie, D. Feng, F. Wang, X. Tang, J. Han, and X. Zhang. Dfpe: Explaining predictive models for disk failure prediction. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 193–204, 2019.

[18] Y. Yi, J. Xiao, S. Wu, H. Li, and H. Jin. Failure order: A missing piece in disk failure processing of data centers. In *2019 IEEE 21st International Conference on High Performance Computing and Communications*, pages 223–230, 2019.

[19] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, 2018.

[20] P. Anantharaman, M. Qiao, and D. Jadav. Large scale predictive analytics for hard disk remaining useful life estimation. In *2018 IEEE International Congress on Big Data (BigData Congress)*, pages 251–254, 2018.

[21] Jiang Xiao, Zhuang Xiong, Song Wu, Yusheng Yi, Hai Jin, and Kan Hu. Disk failure prediction in data centers via online learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 35:1–35:10. ACM, 2018.

[22] C. A. C. Rincón, J. Pâris, R. Vilalta, A. M. K. Cheng, and D. D. E. Long. Disk failure prediction in heterogeneous environments. In *2017 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–7, 2017.

[23] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 391–402. USENIX Association, 2017.

[24] N. Aussel, S. Jaulin, G. Gandon, Y. Petetin, E. Fazli, and S. Chabridon. Predictive models of hard drive failures based on operational data. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 619–625, 2017.

[25] I. C. Chaves, M. R. P. d. Paula, L. G. M. Leite, L. P. Queiroz, J. P. P. Gomes, and J. C. Machado. Banhfap: A bayesian network based failure prediction approach for hard disk drives. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 427–432, 2016.

[26] S. Ganguly, A. Consul, A. Khan, B. Bussone, J. Richards, and A. Miguel. A practical approach to hard disk failure prediction in cloud platforms: Big data model for failure management in datacenters. In *IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 105–116, 2016.

[27] W. Yang, D. Hu, Y. Liu, S. Wang, and T. Jiang. Hard drive failure prediction using big data. In *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*, pages 13–18, 2015.

[28] J. Li, X. Ji, Y. Jia, B. Zhu, G. Wang, Z. Li, and X. Liu. Hard drive failure prediction using classification and regression trees. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 383–394, 2014.

[29] Teerat Pitakrat, André van Hoorn, and Lars Grunske. A comparison of machine learning algorithms for proactive hard disk drive failure detection. In *Proceedings of the 4th International ACM Sigsoft Symposium on Architecting Critical Systems*, ISARCS '13, pages 1–10. ACM, 2013.

[30] Xiaoyi Sun, Krishnendu Chakrabarty, Ruirui Huang, Yiquan Chen, Bing Zhao, Hai Cao, Yinhe Han, Xiaoyao Liang, and Li Jiang. System-level hardware failure prediction using deep learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 20:1–20:6, 2019.

[31] S. Basak, S. Sengupta, and A. Dubey. Mechanisms for integrated feature normalization and remaining useful life estimation using lstms applied to hard-disks. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 208–216, 2019.

[32] F. D. d. S. Lima, G. M. R. Amaral, L. G. d. M. Leite, J. P. P. Gomes, and J. d. C. Machado. Predicting failures in hard drives with lstm networks. In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 222–227, 2017.

[33] S. Pang, Y. Jia, R. Stones, G. Wang, and X. Liu. A combined bayesian network method for predicting drive failure times from smart attributes. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4850–4856, 2016.

[34] C. Xu, G. Wang, X. Liu, D. Guo, and T. Liu. Health status assessment and failure prediction for hard drives with recurrent neural networks. *IEEE Transactions on Computers*, 65(11):3502–3508, 2016.

[35] F. Pereira, D. Teixeira, J. P. Gomes, and J. Machado. Evaluating one-class classifiers for fault detection in hard disk drives. In *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 586–591, 2019.

[36] L. P. Queiroz, F. C. M. Rodrigues, J. P. P. Gomes, F. T. Brito, I. C. Chaves, M. R. P. Paula, M. R. Salvador, and J. C. Machado. A fault detection method for hard disk drives based on mixture of gaussians and nonparametric statistics. *IEEE Transactions on Industrial Informatics*, 13(2):542–550, 2017.

[37] Lucas P. Queiroz, João Paulo Pordeus Gomes, Francisco Caio M. Rodrigues, Felipe T. Brito, Iago C. Chaves, Lucas Goncalves de Moura Leite, and Javam C. Machado. Fault detection in hard disk drives based on a semi parametric model and statistical estimators. *New Generation Computing*, 36:5–19, 2017.

[38] Y. Wang, E. W. M. Ma, T. W. S. Chow, and K. Tsui. A two-step parametric method for failure prediction in hard disk drives. *IEEE Transactions on Industrial Informatics*, 10(1):419–430, 2014.

[39] Y. Wang, K. Tsui, E. W. M. Ma, and M. Pecht. A fusion approach for anomaly detection in hard disk drives. In *Proceedings of the IEEE 2012 Prognostics and System Health Management Conference (PHM-2012 Beijing)*, pages 1–5, 2012.

[40] Xu Zhang, Qingwei Lin, Yong Xu, Si Qin, Hongyu Zhang, Bo Qiao, Yingnong Dang, Xinsheng Yang, Qian Cheng, Murali Chintalapati, Youjiang Wu, Ken Hsieh, Kaixin Sui, Xin Meng, Yaohai Xu, Wenchi Zhang, Furao

Shen, and Dongmei Zhang. Cross-dataset time series anomaly detection for cloud systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1063–1076, Renton, WA, July 2019. USENIX Association.

[41] Y. Xie, D. Feng, F. Wang, X. Zhang, J. Han, and X. Tang. Ome: An optimized modeling engine for disk failure prediction in heterogeneous datacenter. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 561–564, 2018.

[42] F. L. F. Pereira, F. D. d. S. Lima, L. G. d. M. Leite, J. P. P. Gomes, and J. d. C. Machado. Transfer learning for bayesian networks with application on hard disk drives failure prediction. In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 228–233, 2017.

[43] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 39–48. ACM, 2016.

[44] J. Zhang, K. Zhou, P. Huang, X. He, M. Xie, B. Cheng, Y. Ji, and Y. Wang. Minority disk failure prediction based on transfer learning in large data centers of heterogeneous disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2155–2169, 2020.

[45] Charles X. Ling and Chenghui Li. Data mining for direct marketing: Problems and solutions. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, page 73–79. AAAI Press, 1998.

[46] R. Larson. *Elementary Linear Algebra*. Cengage Learning, 2012.

[47] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[48] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[49] Roger Lewis. An introduction to classification and regression tree (cart) analysis. 01 2000.

[50] Vladimir Svetnik, Andy Liaw, Christopher Tong, John Culberson, Robert Sheridan, and Bradley Feuston. Random forest: A classification and regression tool for compound classification and qsar modeling. *Journal of chemical information and computer sciences*, 43:1947–58, 11 2003.

[51] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[52] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[53] Joel Coffman and Alfred C. Weaver. Learning to rank results in relational keyword search. In *CIKM '11*, 2011.

[54] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[55] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[56] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015.

[57] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting disk failures with HMM- and hsmm-based approaches. In *Advances in Data Mining. Applications and Theoretical Aspects, 10th Industrial Conference, ICDM 2010, Berlin, Germany, July 12-14, 2010. Proceedings*, pages 390–404, 2010.

[58] H. Wang, A. Kläser, C. Schmid, and C. Liu. Action recognition by dense trajectories. In *CVPR 2011*, pages 3169–3176, 2011.

[59] Peter Prettenhofer and Benno Stein. Cross-language text classification using structural correspondence learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1118–1127, Uppsala, Sweden, 2010.

[60] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, page 539–546. IEEE Computer Society, 2005.

[61] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging, 2015.

[62] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[63] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[64] Albert Orriols-Puig and Ester Bernadó-Mansilla. Evolutionary rule-based systems for imbalanced data sets. *Soft Comput.*, 13(3):213–225, 2009.

[65] D. Sun, Z. Wu, Y. Wang, Q. Lv, and B. Hu. Risk prediction for imbalanced data in cyber security : A siamese network-based deep learning classification framework. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2019.

[66] Zheng Zhu, Qiang Wang, Bo Li, Wei Wu, Junjie Yan, and Weiming Hu. Distractor-aware siamese networks for visual object tracking. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 103–119, Cham, 2018. Springer International Publishing.

[67] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma. Proactive drive failure prediction for large scale storage systems. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2013.

[68] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227 – 244, 2000.

[70] Ardeshir Raihanian Mashhadi, Willie Cade, and Sara Behdad. Moving towards real-time data-driven quality monitoring: A case study of hard disk drives. *Procedia Manufacturing*, 26:1107 – 1115, 2018. 46th SME North American Manufacturing Research Conference, NAMRC 46, Texas, USA.

[71] B. Eckart, X. Chen, X. He, and S. L. Scott. Failure prediction models for proactive fault tolerance within storage systems. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–8, 2008.

[72] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[73] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, 2017.

Adaptive Placement for In-memory Storage Functions

Ankit Bhardwaj Chinmay Kulkarni Ryan Stutsman
University of Utah

Abstract

Fast networks and the desire for high resource utilization in data centers and the cloud have driven disaggregation. Application compute is separated from storage, but this leads to high overheads when data must move over the network for simple operations on it. Alternatively, systems could allow applications to run application logic within storage via user-defined functions. Unfortunately, this ties provisioning and utilization of storage and compute resources together again.

We present a new approach to executing storage-level functions in an in-memory key-value store that avoids this problem by dynamically deciding where to execute functions over data. Users write storage functions that are *logically* decoupled from storage, but storage servers choose where to run invocations of these functions *physically*. By using a server-internal cost model and observing function execution, servers choose to directly run inexpensive functions, while preferring to execute functions with high CPU-cost at client machines.

We show that with this approach storage servers can reduce network request processing costs, avoid server compute bottlenecks, *and* improve aggregate storage system throughput. We realize our approach on an in-memory key-value store that executes 3.2 million strict serializable user-defined storage functions per second with 100 µs response times. When running a mix of logic from different applications, it provides throughput better than running that logic purely at storage servers (85% more) or purely at clients (10% more). For our workloads, it also reduces latency (up to 2×) and transactional aborts (up to 33%) over pure client-side execution.

1 Introduction

Today, in data centers and the cloud, compute is disaggregated from storage. Separating compute and storage eases provisioning and keeps utilization high by decoupling their allocation. Fast networks have made this practical, but moving all data to compute comes at a cost.

Beyond conventional, higher-level approaches like SQL, many systems have evolved to embed more functionality within storage servers to make storage operations more expressive and to reduce inefficient data movement. For example, some databases allow compile-time extensions [38,47], user-defined functions [34], and stored-procedures [19,22,33,38,48]. Among key-value and object stores, some stores offer a fixed set of extra operators [2,43], while others allow runtime extension with just-in-time [14,26,45] or ahead-of-time compiled user-supplied operations [26]. All of these approaches move user operations closer to the data that they operate on.

The downside is that these approaches fix the ratio of compute to storage, so compute at storage servers can quickly become a bottleneck. The result is that the state-of-practice is to prefer easy provisioning and high utilization while keeping a hard network boundary between compute and storage.

However, the steady decrease in the granularity of compute allocation and scheduling in the cloud (from virtual machines, to containers, to serverless functions) has raised a possibility: application compute need not be statically embedded within storage; nor must it be the case that it is always run separately. Storage servers that support running granular user-supplied functions at low cost create the opportunity to dynamically adapt where functions on stored data are executed. By shifting processing of storage functions back to storage client machines, a storage server can avoid CPU-intensive operations when under load to avoid becoming bottlenecked, choosing instead to send data back to clients for processing. By shifting processing onto itself, a server can eliminate data movement, lend its spare CPU capacity to clients, and reduce its own request processing load. Since moving user-logic into the server reduces the number of requests clients make for data, counter-intuitively, a server can improve its own throughput by taking on more of client applications' compute work.

To show the benefits of such an approach, we developed a new scheme for executing storage functions on top of Splinter [26], which is an extensible in-memory key-value store. Beyond fast `get()`/`put()` key-value operations, applications can push compiled, binary-code extensions containing storage functions to Splinter. These functions can be invoked over the network by clients with low overhead such that even operations that only perform a few microseconds of compute are practical and efficient. Our new approach builds on Splinter to imbue it with a profiler that tracks storage function execution. Clients attempt to invoke their functions at servers. Servers use an internal cost model that weighs the CPU cost to the server if the function were to continue to run at the server against the CPU cost to the server if the function were to run at the client (which would result in extra remote requests to the server for data). Functions invocations that compute over large amounts of data are deemed beneficial and are run at the server, since running them at the client would require transferring large amounts of data. Functions invocations that are compute-intensive but access little data are *pushed back* to the client, where the client must perform the computation.

Beyond the server side, the framework includes a smart storage client library that makes "pushback" cases transparent to applications. The server and the storage client library both provide a binary compatible runtime, so functions are unaware of whether they are run at a storage server (where data access is local) or at a client (where data access is remote). Applications attempt to invoke their storage functions, and the client library transparently executes any client function invocation requests that are pushed back by the storage server before returning the result to the application.

In our model, invocations may execute on the server, at the client, or partially on both, so ensuring consistency is a challenge. Our approach adapts techniques from distributed optimistic concurrency control protocols (OCC) [3, 27, 51] to solve this. All storage functions run within strict serializable transactions, which ensure that clients observe the same strong consistency regardless of where functions execute. These transactions play a key role in the function execution model itself; when a function's execution is transferred from a server to a client, its transaction's read/write set is shipped along with it, avoiding extra requests back to the server for data.

We demonstrate adaptive storage function placement (or *ASFP*) with functions drawn from different domains including aggregation, graph traversal, machine learning classifiers, and authentication. We show these workloads have heterogeneous compute demands, often with compute-to-storage-access ratios varying within one application's functions. Even so, ASFP provides throughput better than running functions purely at storage servers (85% more) or purely at clients (10% more), and it automatically adjusts, optimizing throughput as workloads and server network costs vary and change.

2 Background and Motivation

Today, cloud and data center applications keep data in one set of servers and compute over it on another. This "client-side" function execution model serves as a baseline. Our question is, can a system consistently beat the performance of this client-side approach without creating server bottlenecks?

To do this, one needs a way to embed application logic within storage to compute on data. Our approach relies on the Splinter multi-tenant in-memory key-value store (KVS) [26]. Similar to other low-latency in-memory stores like RAMCloud [40] and FaRM [12], remote clients issue `get()`, `put()`, `multiget()`, and `multiput()` operations to a Splinter server. Unlike most other systems, clients also send compiled *extensions* with custom *storage functions* to it at runtime, which they *invoke* remotely to perform operations over their data. Invoking a storage function only incurs 1,400 cycles of overhead and adds no other no runtime overheads. Splinter achieves low-latency and high-throughput via kernel-bypass networking; one server handles 6.5 million `get()` or 13.5 million no-op `invoke()` requests per second over the network with tens of microseconds of delay. It supports thousands of inter-isolated tenants per server; each application and its storage functions can only access and modify data that it owns.

Extensions reduce requests to storage. With them, a single request could fetch a "user profile" object along with the profiles of friends listed within that profile. Another application could make recursive k-hop queries by traversing edge lists stored in values or make classification requests to stored models. Storage functions access multiple values, reducing server request processing costs, but they are most effective for inter-dependent data accesses since these accesses would otherwise require multiple requests separated by a round-trip.

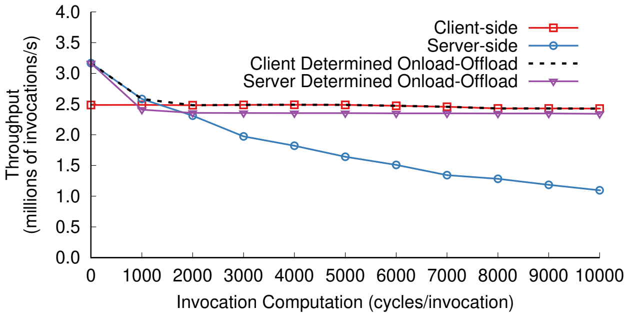


Figure 1: Server-side vs client-side throughput when CPU cost varies. Throughput is inversely related to the compute applied to two accessed values per invocation when run server-side. When the logic is run client-side, the server must process more requests (2 `get()` vs 1 `invoke()` request), but computation is offloaded to clients.

2.1 Understanding the Impact of Placement

Storage functions can lower server overhead, but if functions perform too much computation, then the benefits of eliminating requests are offset as the server CPU becomes a bottleneck. Figure 1 shows this effect. When functions are run server-side (circles), a server performs 3.2 million invocations/second if the functions perform no computation, but its throughput is inversely proportional to the CPU cycles spent computing on the values (x-axis). When run client-side (squares), the server only processes the two `get()` operations for each function; the extra computation is run at clients, which have sufficient idle CPU to perform the work. (Later, we show that if clients do not have idle CPU capacity, our approach shifts work to the server still so long as it is not overloaded. In either case, a global bottleneck is avoided.) When functions perform no computation on values, the two `get()` requests incur higher server-side overhead than sending one `invoke()` request, so server CPU becomes a bottleneck when servicing the equivalent of 2.5 million invocations/second.

Sometimes pure server-side execution provides better system throughput and other times pure client-side execution does. The key insight of this paper is that with lightweight performance tracking, the server can determine this cross-over point, and it can separate invocations into those that should be kept at the server from those that are better run at clients.

2.2 Challenges in Execution Placement

Ideally, a server could get the best of both worlds if it could perfectly determine where to execute an invocation. We simulate this by manually controlling where invocations run based on how much computation they do on data

(`Client Determined Onload-Offload`, dashed line). Here, clients never issue compute-heavy invocations to the server – so, performance matches pure server-side execution for data-intensive invocations and pure client-side execution for compute-intensive invocations.

However, real clients (and real servers) do not know how much computation an invocation will use a priori; different functions vary, and even invocations of the same function could access values and use the CPU in different ratios. Statically determining how much data or how much computation an invocation will use is undecidable in general. Static analysis or modeling might help make good guesses, but the analysis could be fragile and have pathologies.

Our approach is to measure rather than guess; rather than using history, another option is to optimistically assume invocations should run at the server and then try to minimize the cost of correcting mistakes. Figure 1 shows the cost of this conservative, "black box" approach (`Server Determined Onload-Offload`, triangles). Here, clients always invoke functions at the server, but the server quickly sheds invocations that consume CPU without accessing many values. This adds overhead for compute-intensive functions, since the server wastes a small amount of compute before realizing the mistake, but these results show this only hurts throughput 3% for compute-intensive invocations (all other invocations benefit).

Simple enhancements to this scheme are likely to work in practice. Tracking the history of the costs of a particular function's last few invocations can help. If a function's invocations are determined better to be run client-side a few times in a row, then running the next several invocations client-side makes sense. This would work for many applications, but we intentionally avoided such tweaks in this paper. Our approach never relies on the history of invocations (neither across nor within a function); optimizations that make better predictions are likely limited to only recovering that 3% of performance.

In summary, ASFP based on optimistic onloading of application compute to storage with pushback to clients achieves the best of both worlds. For storage functions that access a great deal of data, ASFP avoids data movement costs; for functions that are compute-costly it avoids server bottlenecks.

3 ASFP Design

Applications vary in how they work with data they hold in remote storage. Compute-bound applications may access little data, so moving data to computation is efficient; for data-intensive applications moving computation is more efficient. Multi-tenant stores take this to an extreme: they see a diverse set of applications with a wide variety of compute and data needs. The key idea of ASFP is to exploit this diversity by optimistically colocating functions with the data they access and then profiling storage function execution costs to dynamically relocate invocations that would create a bottleneck.

ASFP relies on *mechanisms* for running storage functions at servers, at clients, or split between both and *policies* to control the mechanisms and decide placement. The four main mechanisms are needed for ASFP are:

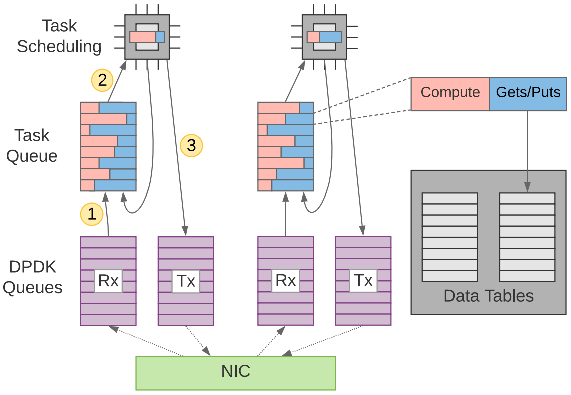


Figure 2: Splinter Request Execution. Each server core has a dedicated network receive queue where clients steer requests. Each core polls this queue and creates a task for each incoming request. Tasks are run round-robin; storage functions can access key-value pairs and perform custom computation on them within the server.

- **Server-side Storage Functions (§3.1.1).** Tenant-provided storage functions reduce data movement. They are key for improving server performance for data-intensive functions. This functionality already pre-exists in Splinter.
- **Server-to-client Pushback (§3.1.2).** ASFP uses a *pushback* scheme that relocates costly function invocations back to clients to avoid server-side bottlenecks.
- **Concurrency Control (§3.1.3).** Since a single function invocation could run partly server-side and partly client-side, consistency becomes an issue. ASFP ensures that this does not cause repeated effects or inconsistencies. It uses OCC to ensure strict serializability of invocation operations, and it integrates with OCC read/write set tracking to preserve work for invocations that are pushed back to clients.
- **Client-side Runtime (§3.1.4).** Clients locally execute invocations that are pushed back from the server, and the ASFP client library makes this transparent. Applications wait for invocations to complete; the client library runs pushed back invocations, fetching data from the server as needed.

The server's primary objective in ASFP is to minimize the CPU usage per function invocation and to optimize its own throughput, which, indirectly optimizes the throughput of the entire system. The ASFP policy relies on three key components to do this:

- **Invocation Profiling.** Each server tracks each function invocation as it runs to account for its CPU time.
- **Request/Response Cost Modeling (§3.2.1).** Similarly, each server dynamically profiles networking CPU costs to determine a CPU cost model for data movement. This projects how much server CPU is being saved by running each invocation at the server. If an invocation has consumed substantially more CPU cycles at the server than the request/response cost model projects have been saved by running it at the server, then it is pushed back to the client.

Overload Trigger (§3.2.2). Even compute-bound functions run more efficiently at the server than they do at clients since they can avoid data movement. All invocations run at the server if there is spare CPU capacity available, so long as they don't create a bottleneck at the server. Hence, pushback is only triggered when our server deems itself overloaded.

First, we describe ASFP's mechanisms to show how storage functions, pushback, and concurrency control work; then, we explain how its measurements and policies drive its mechanisms. Overall, ASFP constitutes about 7,500 lines of code split across additions to the Splinter server and a the new client library, which shares much of its code with the server (available at `https://github.com/utah-scs/splinter/`).

3.1 ASFP Mechanisms

3.1.1 Server-side Storage Functions

ASFP is built on top of the Splinter in-memory KVS. Splinter supports typical KVS remote `get()` and `put()` operations. It is a good starting point because it also supports installation of client-supplied native-code extensions. These extensions add storage functions to the server that can be called remotely via `invoke()` requests. ASFP uses `invoke()` requests to move computation to data, and it extends Splinter with new profiling, policy, and function invocation relocation functionality.

Internally, Splinter multitasks between `get()`, `put()`, and (possibly longer-running) `invoke()` requests, so each incoming request is converted into a *task*. The server runs these cooperative tasks round-robin until they complete or yield; this prevents head-of-line blocking when functions take awhile to execute. Tasks handling `invoke()` operations maintain state for the running storage function as a coroutine stored in the task. Figure 2 illustrates request processing. Each server core polls a CPU-core-specific network receive queue and creates a task for each incoming request (①), each of which is added to a per-core task queue. Each queued task is run once (②), then the core polls its receive queue again. The core transmits a response (③) when a task completes and then destroys it.

Invocations run interleaved due to cooperative scheduling, but they can also run in parallel too. Clients steer requests to specific CPU cores to reduce overhead, but server cores steal work from each others' receive queues to keep throughput high under load imbalance. Hence, pipelined invocations from a client can run in parallel at the server.

3.1.2 Pushing `invoke()`s Back to Clients

ASFP lets Splinter servers selectively shed load. When a storage server's cores are overloaded, it *may* perform a *pushback* on tasks. These tasks are terminated at the server and restarted client-side. Figure 4 shows the state transition diagram of the lifecycle of an `invoke()` request at a server. ASFP adds a new `Offload` state to server-side tasks to support pushback.

For each incoming `invoke()` request, a server creates a task and tries to run it to completion, sending a `Result` response (top of Figure 3). However, if a server is past an *overload trigger point* (§3.2.2), then it chooses some `Ready` `invoke()` tasks that are good candidates for client-side execution based on a *threshold function* (§3.2.1), and it moves them to `Offload`.

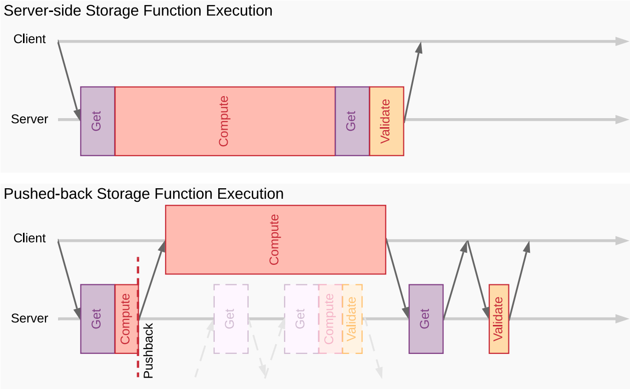


Figure 3: Timeline of a function invocation request when run server-side (top) and when pushed back to the client side (bottom). In this case, offloading this relatively long-running function to the client gives the server extra CPU resources to service other requests.

When tasks in the `Offload` state are scheduled, a `Pushback` response is generated that informs the client that it should run the function client-side. The client runs the function, falling back to making `get()` requests to the server to fetch needed values (`put()`s are cached locally and installed atomically when the invocation completes, §3.1.3). Figure 5 shows this. If the client receives a `Result` response, the work of the requested invocation has been done, and there is nothing left to do. If the client receives a `Pushback` response, the client begins to execute the function logic itself in a fashion similar to the server. The bottom of Figure 3 shows the interactions between the server and the client when an invocation is pushed back to the client; as shown, this avoids a bottleneck in this case, freeing the server to process other requests at the server.

3.1.3 Consistency and Concurrency Control

Storage functions and pushback create interrelated challenges, especially for consistency. First, `invoke()` tasks run concurrently at the server; this can happen because tasks run interleaved at the server and because server cores perform work stealing. Pushed back requests also create concurrency, since those functions run at the client in parallel with server tasks. Second, when tasks are pushed back, the client restarts execution of that function from the beginning – pushback has no means to preserve the running state of a function to resume it at the client. This means that without care, clients might repeat operations, which would affect the concurrent behavior of functions and make it hard to reason about consistency.

To solve these consistency issues, invocations are run as strict serializable transactions. This makes it easy to reason about consistency regardless of where an invocation is run. ASFP adds OCC transactions to Splinter. When a server receives an `invoke()`, it creates an empty read/write set. The server tracks the version of each value that a task sees and the values that the tasks wishes to install in storage. If the task

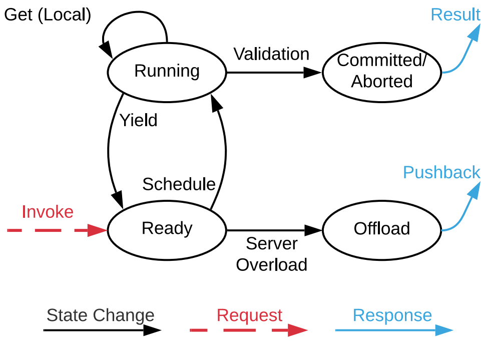


Figure 4: Server-side task states for an `invoke()`.

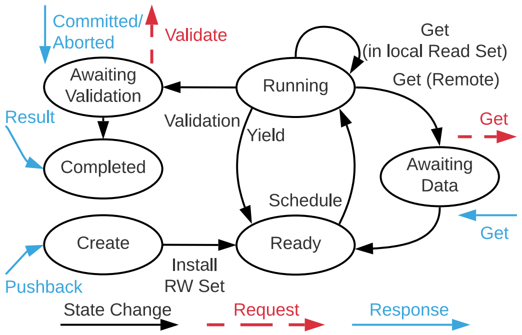


Figure 5: Client-side task states for a pushed-back `invoke()`.

completes on the server, validation is performed by latching the invocation's read/write set. For each key in the read set, if the associated value versions remain unchanged, then its write set is installed and the client is informed of commitment; otherwise, the server indicates abort.

The server's read/write set has a second purpose: by tracking what values an invocation has read, the server can save work by returning those values immediately on pushback. The client installs this read/write set locally before restarting the function. This way, the server will never have to repeat any work for a pushed back task: all of the values the task needs have already been delivered to it up to the point that it was terminated at the server. This is key: pushed back requests never generate extra work for the server. This bears similarities to reconnaissance queries in deterministic databases [50].

On completion of a pushed back task, the client sends the write set and version metadata for the values it read to the server where validation is performed the same as if the task has completed server-side. This is another advantage of OCC: the server need not keep any state about an invocation once it is pushed back. For example, the server retains no metadata or locks on behalf of a pushed back task. This makes any recovery or state reclamation unnecessary on client failures.

3.1.4 Client Runtime for `invoke()`s

Splinter client requests consist of basic `get()/put()` requests and `invoke()` requests that attempt to invoke a storage function within the store. Clients register *extensions* at the server before invoking the functions they contain. ASFP requires that the same extensions are registered at the client library as well, so that they can handle pushed back `invoke()` requests.

On each `invoke()` response from the server, the client checks a `Pushback` response flag. If it is set, the client performs logic similar to request dispatching on the server: it creates a task and coroutine similar to the ones used on the server, and it places the task in a client-local task queue (Figure 5). The main difference is that the read/write set returned from the server is used to pre-populate the read/write set of the invocation before it starts at the client side.

Clients put `Ready` tasks in their task queue and run tasks round-robin, just like the server. This lets clients make progress on invocations while continuing to issue new operations to the server.

The ASFP client library provides a binary-compatible interface with the server, so identical versions of storage functions work whether they run at the client or at the server. Splinter extensions have a restricted `get()/put()` interface for interacting with storage, and they have a restricted set of whitelisted library functions they can run beyond that. Extensions on the client-side have the same restrictions, so that pushed back invocations will run the same way in both places.

Client-side execution does run different from server-side in one important regard: the client must access key-value pairs remotely. This is solved by passing in handles into storage functions through which they request access to data. On the server-side the handles call `get()/put()` functions to access data directly; on the client-side the handles issue remote `get()/put()` requests. Requests to the store take about 10 µs to service, and task context switch time is just 24 cycles, so tasks waiting for responses from the server enter an `Awaiting Data` state. Each invocation has a unique client-side id; whenever the client library receives a response to a particular extension invocation, it adds the record to the local read/write set for that invocation. Clients read through their read set; after a "hit" in their read set or upon the completion of a remote access, the task is returned to the `Ready` state.

3.2 ASFP Policies

3.2.1 `invoke()`s Profiling and Classification

Splinter is both multi-tenant and extensible. Together, these mean that it must deal with different access patterns and functions with varying compute-to-storage-access ratios. This also means a server will be able to find many suitable functions to run that can reduce its load. When overloaded, it must determine which `Ready` tasks should be pushed back; however, pushing back the wrong tasks can *hurt* its throughput.

Whether a task is beneficial to server throughput when run server-side is determined by two things: the amount of CPU time it uses computing on the values it accesses (which hurts throughput) and the number of values it interacts with (which benefits throughput, since each access run at the server eliminates a network request that it otherwise would have had to process). Effectively, each time a task accesses a stored value it should be credited for the amount of server CPU it saved by having run that operation locally. Likewise, whenever it performs other computation that does not save server CPU work it should be debited, since this slows request processing.

This results in a natural threshold for when tasks would be pushed back to clients, which we call the *pushback threshold*. It is defined by

$$c < nD - (D+I)$$

where c is the amount of computation done by an invocation so far, n is the number of values accessed by the invocation so far, D is the request processing CPU cost, I represents the cost to perform an invocation (beyond request processing cost). Effectively, nD is the work the server would have done if the client issued n `get()` requests. $(D+I)$ represents CPU cost at the server of an `invoke()` request. Hence, so long as $c < nD - (D+I)$, server-side work is saved by letting the invocation remain at the server.

This inequality divides all tasks into two classes, $\mathcal{S}$ and $\mathcal{C}$. Tasks in $\mathcal{S}$ are beneficial to run at the server, and tasks in $\mathcal{C}$ improve server throughput when run at clients. The inequality does not hold when an invocation accesses zero or one values; these invocations save the server less work than the cost of an `invoke()` request. It never makes sense to run them at the server, and it would also be unusual for a client to try to do so. The model is simple and linear, so the server can calibrate it inexpensively at runtime. Just by profiling the cost of an `invoke()` operation and a `get()` operation, it can accurately assess which invocations should be pushed back.

As discussed in Section 2.2, it is undecidable in general to determine the class of an invocation. The input parameters to an `invoke()`, the data its accesses, the server hardware, and its cache policies/pollution all influence performance. Our approach simply assumes all invocations should be initially attempted server-side. Exploiting history or domain knowledge would improve performance in cases where functions are pushed back. However, we explicitly avoid relying on such information since its effectiveness is workload dependent, and it can only provide a few percent performance improvement for invocations in $\mathcal{C}$ (and would only hurt functions in $\mathcal{S}$).

3.2.2 Server Overload

The final piece of ASFP is overload detection. Functions should always run at the server when it has idle CPU; this still eliminates data movement costs, and it frees client CPUs to do other work. However, when overloaded, the server must shed load to improve throughput and control response times.

Algorithm 1 shows how the server detects overload (others use similar approaches [39]). The server is under high load when it receives new tasks and the requests from previous scheduling passes have not completed. At the beginning of a server's round-robin pass through its set of tasks, it polls its receive queues and creates up to B tasks, one for each incoming request (where B is the maximum receive batch size, which we fix at 32). Then, it compares the amount of time spent dispatching requests in that round of scheduling (time spent polling network queues and creating tasks) with the time spent executing invocation tasks in that scheduling pass. If invocation task execution time is the dominating factor, the scheduler checks the task queue length. If it contains at least B/k tasks and processing incoming requests would create at least another B/k tasks, then the scheduler sets a flag indicating the server is overloaded. Higher values of k trigger overload more easily; $k = 2$ works well, and we keep it fixed in our experiments. This guarantees that the scheduler:

1. only pushes back work if load is mainly from `invoke()`s;
2. keeps at least B/k tasks in the queue after pushback; and
3. only pushes back when $\geq 2B/k$ requests await service.

On overload, the server tests the threshold inequality (§3.2.1) on all old `Ready` tasks, triggering pushback on some of them.

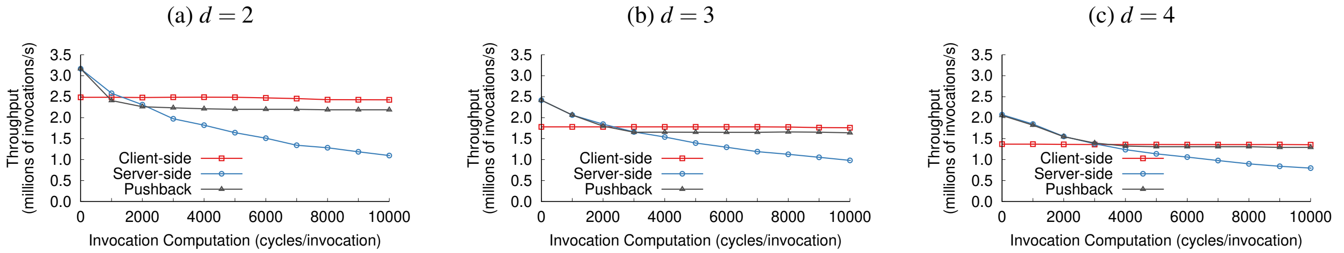
Algorithm 1: Server Overload Detection

```
Function Scheduler()
    totalTime ← 0;
    while true do
        t ← taskQueue.Dequeue();
        if t = DispatchTask then
            newTasks, dispatchTime ← PollRecvQueue();
            if totalTime ≫ dispatchTime then
                if taskQueue.length ≥ B/k and
                  newTasks.length ≥ B/k then
                    taskQueue.ClassifyAndPushback();
                    taskQueue.Enqueue(newTasks);
                end
                taskQueue.Enqueue(t);
                totalTime ← 0;
            end
        else if t = RequestTask then
            t.getPutTime, t.computeTime, t.state = t.Run();
            totalTime += (t.getPutTime + t.computeTime);
            if t.state ∉ {Committed, Aborted} then
                taskQueue.Enqueue(t);
            end
        end
    end
```

4 Evaluation

We compare three models for storage functions. *Client-side* runs them on clients and issues `get()` requests to the server. This is state-of-practice and the baseline. *Server-side* runs functions on the server and represents Splinter's approach. *Pushback* is our approach, which runs functions on the server, pushing some back to clients. We focused on these questions:

Does ASFP improve storage server throughput? For an application mix consisting of machine learning classification and graph-based storage functions, ASFP can improve throughput by 10% (§4.5.3). For functions with dependent data accesses, ASFP improves throughput by 42% (§4.2).


(a) $d = 2$

(b) $d = 3$

(c) $d = 4$

Figure 6: A function with 2, 3, or 4 dependent `get()`s followed by varied computation lengths. ASFP improves throughput by 42% over client-side execution. For compute-intensive invocations, ASFP throughput is within 15% of pure client-side in the worst case.

What is the cost of using ASFP? For invocations that improve server throughput (those in class $\mathcal{S}$), ASFP gives the full performance benefit of server-side execution with no measurable overhead. For compute-intensive invocations that access little data, classification and client-side re-execution delivers performance within 15% (§4.2) of pure client-side execution in the worst case.

How effective is the ASFP classifier? For a mix of predominantly $\mathcal{C}$-class invocations with significant compute variance, 87% of all $\mathcal{C}$-class invocations are offloaded to clients (§4.3). The remaining are accurately classified but retained by the server because it has idle compute to execute them, improving overall system throughput.

How does ASFP impact latency? For invocations in class $\mathcal{S}$, ASFP saves on round trips to the server, which reduces latency by as much as 2× (§4.4). For invocations in class $\mathcal{C}$, ASFP's read/write set and server overload-based optimizations can help reduce latency by 15% compared to executing client-side. For extremely compute-intensive invocations, ASFP matches client-side execution.

How do ASFP and OCC interact? Beyond providing consistency, OCC lets the server send back read/write sets on pushback, improving throughput by 33% (§4.6). ASFP also exploits idle compute at both servers and clients speeding up transactions and reducing abort rates (§4.6.1).

4.1 Experimental Setup

Evaluation is on five machines; four as clients and one server (unless otherwise noted) on CloudLab [13] (Table 1). All use DPDK [11] over Ethernet. Eight of ten server cores process requests; Splinter uses two cores for task management. Clients also use eight cores; each core pipelines `invoke()` requests up to a depth of 32 and receives responses in a closed-loop.

Using a closed-loop is helpful. ASFP demands complex, heterogeneous workloads; an open-loop load requires careful manual pacing of the request rate for each storage function type. To ensure we always measure the server at saturation (unless otherwise noted), we control client thread count instead of manually tuning per-storage-function request rates.

The server held 15 GB as 120 M records (30 B keys, 100 B values) unless otherwise noted. On pushback, clients transparently ran functions locally, issuing remote record requests.

CPU	Ten-core Intel E5-2640v4 at 2.4 GHz
RAM	64GB Memory (4x 16 GB DDR4-2400 DIMMs)
NIC	Mellanox CX-4, 25 Gbps Ethernet
Switch	Mellanox SN2410 48-port, 25 Gbps per port
OS	Ubuntu 16.04, Linux 4.4.0-138, DPDK 17.08 Rust 1.29.0-nightly, 16×1 GB Hugepages

Table 1: Experimental setup. Evaluation used one machine as a server and four as clients. All experiments were run on CloudLab.

4.2 ASFP Throughput Benefits & Costs

Benefits. ASFP combines the benefits of server- and client-side execution; invocations with low compute-to-data access ratios run on overloaded servers, otherwise they are offloaded to clients. To show this, we run a microbenchmark that varies the number of records accessed and the amount of compute performed within an invocation.

Clients issue `invoke()`s that do d data-dependent `get()`s followed by x cycles of compute. Figure 6 shows server throughput when the function is run purely client-side, purely server-side, and with adaptive pushback for d from 2 to 4. With a small x, server-side execution prevents clients from stalling on remote `get()`s. With a large x, client-side execution with remote value access avoids a server CPU bottleneck.

Here, ASFP's simple model works well. In Figure 6 (a), invocations that perform little compute over values stay at the server, improving throughput over client-side execution by 27%. Invocations using more CPU are pushed back, and throughput tracks the client-side approach. For increasingly CPU-intensive invocations ($x >$ 6,000), the throughput of pure server-side execution tends toward zero, so the benefits of pushback over server-side execution grow (until all client CPUs saturate, but realistic servers will service many clients).

These results show that the more data an invocation accesses, the more savings pushback provides; increasing d to 3 and 4 gives savings of up to 33% and 42%, respectively (Figure 6 (b), (c)). The area between pushback and client-side can be large for CPU-inexpensive functions (left side of graphs), but the area between pushback and server-side for CPU-expensive functions (right side) is also large since real functions will vary even more in how much CPU they use.

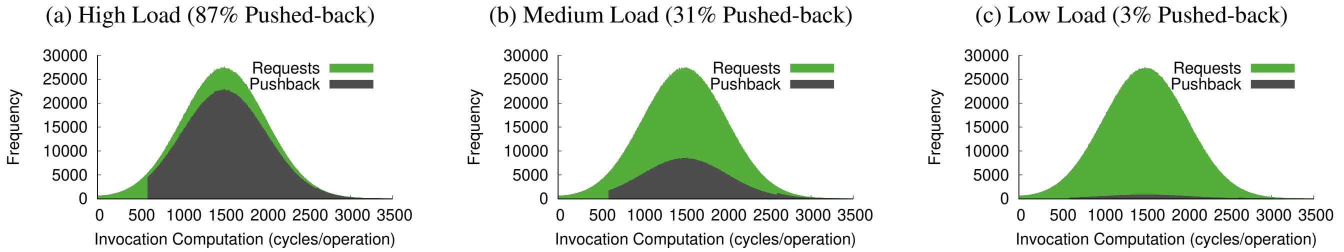

(a) High Load (87% Pushed-back) (b) Medium Load (31% Pushed-back) (c) Low Load (3% Pushed-back)

Figure 7: Distribution of invocations generated overlaid with those pushed back to one client under different loads. ASFP can use idle server compute to run some of the invocations classified as $\mathcal{C}$. This is why the distributions do not completely overlap.

Costs. These graphs also show ASFP's costs. On a pushback, there are two costs: the first is the cost of having the server process an extra request for `invoke()` and validation; the second is the computation the invocation did before it was terminated. Most of the first cost is eliminated by shipping all accessed values back to the client on pushback. An `invoke()` only costs 9% more than a `get()`, and all practical functions receive some values when they are pushed back. Hence, the cost of the `invoke()` is offset by the fact that it eliminates the need for the client to issue at least one `get()`.

The second cost explains the gap between the client-side and the pushback approach. This experiment is a pessimistic case: the function first accesses all of its values, then it performs compute over those values. For compute-intensive functions, this means the server runs them longer before it pushes them back; for functions where data access and computation are intermingled, pushback would achieve performance closer to the client-side case. Even so, in all cases where client-side execution would outperform server-side, pushback is only 13 to 15% slower than running everything at clients.

Cost Breakdown. This 15% overhead for ASFP for $\mathcal{C}$-class functions in Figure 6 (a) has two components. The first is the cost of suspending an invocation and sending its read/write set back to the client, but this only accounts for 3% of the 15%. Figure 1 shows this; in it, the performance difference between (omniscient) client-determined placement and server-determined placement that observes each invocation is only 3% even when invocations are in $\mathcal{C}$. The second component of the overhead (12%) comes from an interplay between overload detection and $\mathcal{C}$-class invocations. In Figure 1, the server never executes an invocation in $\mathcal{C}$ to completion, even if the server is idle, but ASFP completes invocations server-side, regardless of class, if the server is underloaded. However, a server's load can shift rapidly at fine timescales. Leaving $\mathcal{C}$-class invocations onloaded is a form of speculation about whether invocations will arrive in the near-term that will overload the server. This 12% is due to cases where the server performed some $\mathcal{C}$ work, and it became overloaded during that work. This effect can be seen in §4.3 Figure 7 (a) as well; even at high load some $\mathcal{C}$-class functions are run at the server. This can be controlled; making overload detection more aggressive reduces this overhead (down to 3%, if desired); the trade-off is that the server may sit idle in more cases to ensure it has capacity when $\mathcal{S}$-class tasks arrive.

4.3 Invocation Heterogeneity

Real invocations are likely to be heterogeneous in two ways: first, the total compute performed might vary across invocations (*inter-invocation heterogeneity*), and second, compute might be clustered at points of execution instead of being evenly distributed across data accesses (*intra-invocation heterogeneity*). To be effective, ASFP must be able to accurately classify invocations (as $\mathcal{S}$ or $\mathcal{C}$), as well as efficiently use both server and client CPU under such forms of heterogeneity.

Inter-Invocation Heterogeneity. To demonstrate ASFP's efficiency under inter-invocation heterogeneity, we configured one client to generate `invoke()`s where the number of cycles of extra compute performed (after two dependent `get()`s) is drawn from a normal distribution, $\mathcal{N}(1500, 500)$. Figure 7 (a) plots the distribution of the generated requests overlaid with the distribution of those that were pushed back and completed on the client. This figure shows two things; first, no requests that perform less than 600 cycles of work are pushed back, so inexpensive functions are executed at the server; second, the two distributions do not completely overlap, so many compute-intensive invocations still complete at the server. With just one client, the server has some idle CPU capacity; as a result, many of invocations in $\mathcal{C}$ run server-side. As the load on the server decreases, this spare capacity increases, allowing more $\mathcal{C}$ invocations to run server-side (Figure 7 (b), (c)). This shows that ASFP can be efficiently split work between the server and client(s); any idle compute at storage can accelerate clients and improve throughput.

Intra-invocation Heterogeneity. Figure 6 presented the benefits and costs of ASFP when compute is performed after all records are accessed by an invocation. Under this scenario, pushed back invocations benefit from the shipped back read/write set. However, real function invocations are likely to perform compute at different points of execution (between record accesses for example). Figure 8 explores such scenarios. 'Pushback-*y*' represents a run where invocations perform compute after issuing *y* `get()`s (out of a total of 4 per

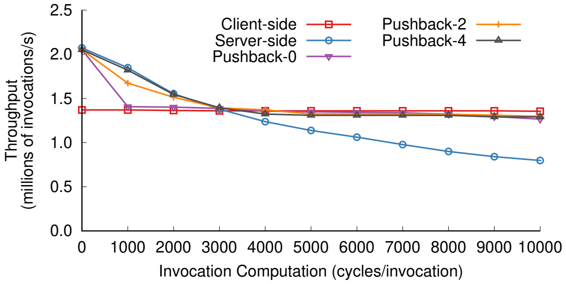


Figure 8: Throughput when the position of compute within an invocation varies. 'Pushback-*y*' is when invocations perform compute after issuing *y* `get()`s. ASFP is never worse than pure client-side.

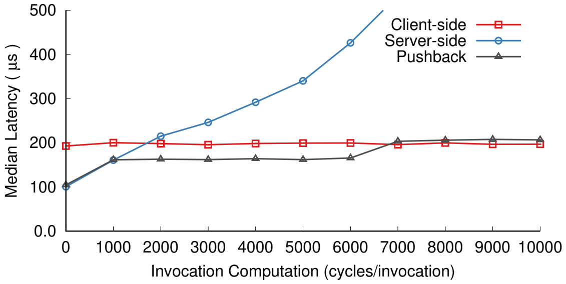


Figure 9: Effect of ASFP on median latency. ASFP improves latency between 15% to 2× compared to pure client-side.

`invoke()`). ASFP throughput is always better than or equal to pure client-side execution; for cases where compute is performed early on (Pushback-0), compute inexpensive invocations (left side of the graph) get pushed back earlier, resulting in lower gains over pure client-side execution.

4.4 ASFP Impact on Latency

Figure 9 shows median latency for an experimental setup similar to Figure 6 (a). When compute is less than 600 cycles, ASFP reduces round trips by running invocations on the server, improving latency over pure client-side execution by as much as 2×. As compute increases, invocations get pushed back; ASFP's latency is still better (15%) because these invocations receive their read/write set, resulting in one less RPC compared to client-side execution. The pure server-side approach bottlenecks, causing its response times to spike. As compute increases beyond 6,000 cycles, pushed-back invocations cause clients to saturate, reducing server load. This makes ASFP's overload detection loop retain more invocations on the server, increasing median latency to track that of pure client-side execution. Since, ASFP restarts pushed-back invocations at clients, it can increase the latency by up to 2× in the worst case. However, the only invocations that could experience this worst case are ones that never access values, since invocations that access values always execute more quickly on the client-side after pushback due to read/write set shipping. These functions should be rare and should not be attempted at storage servers; clients have little reason to send them to the server since they never access data.

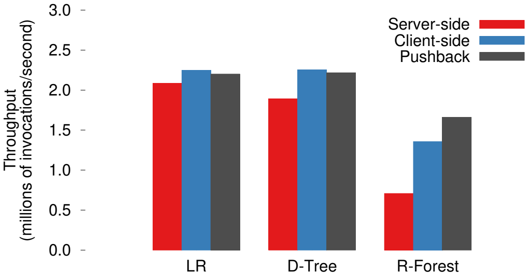


Figure 10: Machine learning classifiers. LR and D-Tree are within 2% of pure client-side. For R-Forest, ASFP leverages idle server compute, improving performance by 22% over pure client-side.

4.5 Realistic Applications

Beyond microbenchmarks, we applied ASFP to more realistic functions. We use three types of functions from different domains that we believe would be a possible fit for low-latency in-memory storage services. The first is an application barely in class $\mathcal{S}$ that accesses little data and performs little compute per invocation: Facebook's TAO social graph database [5]. The second is an application barely in class $\mathcal{C}$ that accesses little data with compute requirements slightly higher than the $\mathcal{S}/\mathcal{C}$ threshold: a machine-learning based disk failure prediction. The last is an application well in class $\mathcal{C}$ that accesses little data and uses significant CPU: authentication [42]. To be effective, ASFP must classify (as $\mathcal{S}$ or $\mathcal{C}$) and place invocations (on the server or the client) and improve overall throughput. We show ASFP can do so for these functions and for mixes of both $\mathcal{S}$ and $\mathcal{C}$ invocations.

4.5.1 Machine Learning

We use disk failure prediction [17, 28, 41, 44] for our first application. This application consults a classifier to predict whether a disk in a data center is about to fail. We chose classifiers because they benefit from Splinter's model that supports complex but native functions; they are a realistic and expected use; and their compute requirements vary.

We evaluated three classifiers: logistic regression (LR), a decision tree (D-Tree) and a random forest (R-Forest) (an ensemble of decision trees). Classifiers are trained offline from a data set with 25 features [37]. The server holds data points to be classified (loaded/streamed in a priori). Two clients generate `invoke()`s that classify two data points each.

Figure 10 shows how the classifiers perform. All three are in $\mathcal{C}$, so client-side execution outperforms server-side. R-Forest is the most CPU-intensive. ASFP outperforms both pure client-side (22%) and pure server-side (2.3×) execution because invocations are placed on both the server and the client. LR and D-Tree are harder cases; they are nearer to the $\mathcal{S}/\mathcal{C}$ split; the extra overhead of initially running them server-side before pushing them to the client cuts into ASFP's benefits. As a result, for these two classifiers, pure client-side execution marginally outperforms ASFP (by $<2\%$) even considering the extra compute capacity that the server provides.

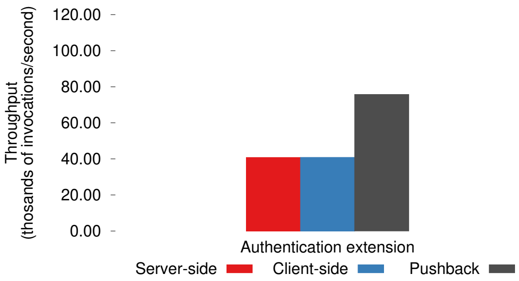


Figure 11: Authentication application. Compared to running server-side or client-side, ASFP can exploit idle cycles anywhere among the machines, improving throughput by nearly 2×.

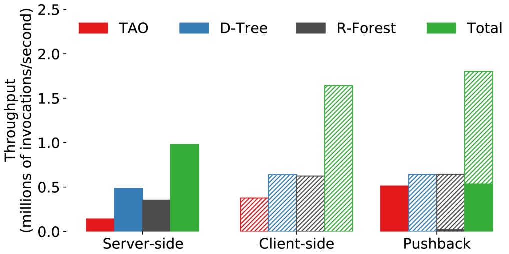


Figure 12: TAO/D-Tree/R-Forest Mix. ASFP correctly places invocations, improving overall throughput by 10%. Solid-colored regions of the bars show throughput due to invocations that ran server-side; hashed regions show throughput due to those that ran client-side.

4.5.2 Authentication

Authentication is another application; it uses bcrypt [42] to verify user identity. It uses few values, and it is computationally costly (well in class $\mathcal{C}$). Even so, it can still benefit from ASFP. We ran an experiment over 128,000 records, each containing a 30 B username and a 40 B salted hash (16 B salt, 24 B hash). One client issues `invoke()`s with a username and a 72 B AES-encrypted password. The salted hash is applied to the password. If the result matches the stored salted hash, then the invocation returns success, otherwise it returns failure.

Figure 11 shows throughput. Purely server- and client-side execution perform about 40,000 authentications/s. Both are CPU bottlenecked; bcrypt takes about 450,000 cycles per request. With pushback, throughput is nearly doubled over both approaches, as expected; with ASFP, CPUs on both the server and the client can be used to perform authentication.

4.5.3 Application Mix

Splinter is expected to run multi-tenant workloads, and pushback is primarily beneficial in a setting where there are a wide and heterogeneous set of `invoke()` requests. To create such a scenario, we ran a mixed workload comprised of an R-Forest classifier (class $\mathcal{C}$), a D-Tree classifier (class $\mathcal{C}$ by a small margin), and an implementation of Facebook's TAO [5, 26] data model which consists of dependent data accesses (class $\mathcal{S}$). Three client machines generated requests to the server.

Figure 12 shows how ASFP improves throughput for this mix. The solid-colored regions of the bars show throughput achieved from running invocations server-side. The hashed regions of bars show throughput achieved from running invocations client-side. The final bar in each group shows the aggregate throughput of the three applications (both at clients and the server). R-Forest requests are CPU-intensive; so, they are bottlenecked by server CPU in pure server-side execution, *and* they hurt the throughput of the other applications sharing the server. Running functions client-side avoids this bottleneck and interference, raising the throughput of the other applications. However, this runs TAO at clients as well, which creates extra server load since it is in class $\mathcal{S}$. Hence, ASFP provides the best results. R-Forest and D-Tree are classified as $\mathcal{C}$ and run (almost completely) at clients. TAO is classified as $\mathcal{S}$ and runs at the server improving server throughput. Hence, ASFP provides 10% better throughput than a conventional, disaggregated approach. Interestingly, onloading TAO creates CPU headroom at the server that R-Forest is able to exploit.

This workload is a challenging one for ASFP; a majority of the TAO requests only access one data item per invocation (60%); hence, all of the applications perform fairly well when executed client-side. As a result, the system only experiences modest gains when TAO is run at the server.

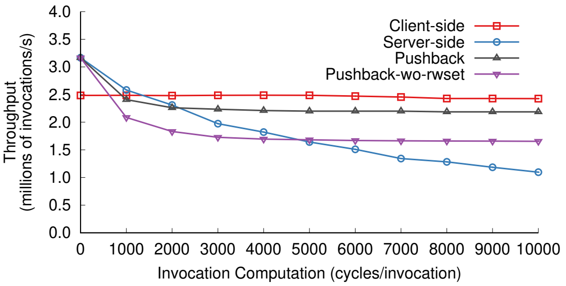


Figure 13: Impact of read/write set shipping on ASFP. When turned off, throughput suffers by 33% since pushed-back invocations load the server by reissuing remote `get()` requests.

4.6 Concurrency Control and ASFP

Beyond providing consistency, OCC improves ASFP's throughput; instead of reissuing `get()` requests to the server and increasing its request processing load, pushed-back requests reuse their read/write set. We explore this optimization with a setup similar to Figure 6 (a). Figure 13 shows that disabling read/write set shipping for pushed back invocations hurts throughput by 33% (Pushback-wo-rwset). Note, this workload only accesses two records per invocation; invocations that access more records would benefit more.

4.6.1 ASFP Impact on Abort Rate

Moving execution between servers and clients affects transaction commit latency and abort rates. To study this, we used YCSB+T's *Closed Economy Workload* [9] with four clients generating a request distribution where 50% of the requests are read-only and the remaining are read-modify-writes. We added a parameter to the read-modify-write transactions to control how much compute each one performs.

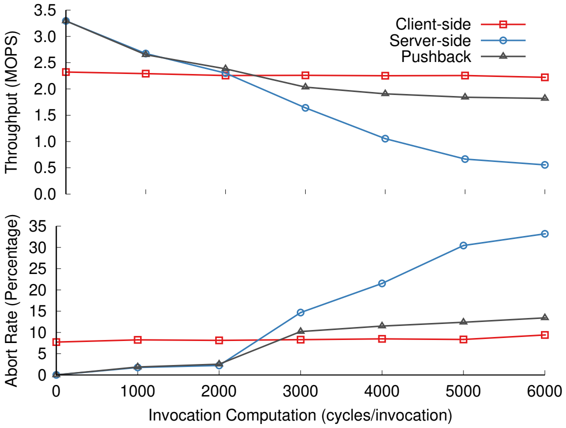


Figure 14: Impact on abort rate. ASFP leverages idle compute to speed up transactions, reducing read/write conflicts and abort rates.

Figure 14 shows trends similar to §4.2. ASFP speeds transactions/invocations in $\mathcal{S}$ by placing them on the server, reducing read/write conflicts and aborts. Server-side execution would bottleneck and slow transactions/invocations in $\mathcal{C}$, increasing conflicts. Here, ASFP uses clients to speed invocations, reducing aborts. The conflict window of each transaction is mainly determined by its latency. This relationship can be clearly seen when comparing these results to those in Figure 9. ASFP avoids bottlenecks, controls latency, and reduces aborts regardless of how much compute invocations use. We omitted results for a 90-10 read/write ratio; aborts are always negligible (0.02%), even for invocations in $\mathcal{C}$.

5 Discussion

Security. ASFP builds on Splinter's unique function isolation model that uses Rust's type system. This software-based scheme has a broad attack surface including Splinter's code; Rust (its type system, compiler, and standard library comprising millions of lines of code); and underlying libraries including libc and DPDK. This model is also complicated by micro-architectural side channels and speculative execution attacks, which continue to surface. For example, Splinter does not include the micro-architectural state flushes needed on protection domain switches to protect against information leaking via Spectre v2 and other similar vulnerabilities [6, 23].

ASFP is independent of Splinter's isolation and trust model, but there two ways that its isolation costs affect ASFP's applicability to other systems. First, its software-based isolation has extremely low protection domain/context switch costs. Tenant function invocations cause neither page table nor stack switches; hence, `invoke()`s are only 9% more expensive than `get()`s. With stronger isolation schemes, like conventional page table switching, each `invoke()` would need to make up for these costs, which can add up to several microseconds of CPU. With Splinter, some functions that only access two records improve efficiency when run server-side; if a page table switch were needed per invocation, invocations that accessed less than tens of records would be inefficient server-side. The second impact of Splinter's model is that it supports thousands of tenants per machine with low overhead, increasing the heterogeneity of operations it would be offered by tenants. Overall, this means using stronger isolation primitives would result in providers dedicating at least one server core to each tenant to avoid protection domain switch costs; this would limit the diversity of functions each server handles.

Larger-than-DRAM data & distribution. ASFP targets low-latency in-memory storage where only small, hot records are economical to store, which simplifies its cost model. Records are so small that the CPU cost of copying them (in/out of network buffers) is negligible, and neither I/O CPU cost nor storage throughput limits need to be considered. Addressing more complex systems is an interesting problem.

ASFP is focused on one server and its clients. As-is it can work in a sharded store where data is partitioned (e.g. by key). In the future, we plan to extend its OCC model for distributed transactions while factoring in data movement costs and abort rates in deciding placement of operations.

Idle client-side CPU assumptions. ASFP assumes clients have sufficient idle CPU to run pushed back invocations. This relies on provisioning client capacity according to state-of-practice: as if all invocations run client-side. When invocations are shifted server-side, this can only produce extra idle capacity at clients and servers.

State migration. A full system would need sharding, load balancing (similar to Slicer [4]), state migration to consolidate load [25], and a means to deprovision idle CPUs. ASFP is complementary; load and state must be rebalanced in any cluster with or without ASFP. For the heterogeneous invocations offered to servers, ASFP optimizes server CPU regardless of how state is sharded across the cluster.

Restart vs. resume. Process/function migration [10, 32, 35] is costly and complex. Resumed functions would need to send intermediate state to clients; that additional state capture, transmission, and restoration would need to be incorporated into ASFP's cost model. Further, restarted, pushed back functions take no more client-side CPU than they would in today's client-side approaches. Worst case, a function could be (nearly) computed at the server and repeated at a client. In the cases we have looked at redundant work is small, so it would be hard to offset function shipping/resuming costs.

Predicting placement. Speculatively onloading a function only adds 3% server load even when it is always pushed back (Figure 1). Pathological cases could access many records after pushback. These would perform nearly the same as today's pure client-side approach, but history/prediction could help. Simple approaches that track recent invocation misclassifications could be used to bias a function's future invocations to stay server-side. Pushback only happens on overload, so some misclassification has little impact; the server need only classify enough tasks correctly to mitigate overload.

Other key-value stores. ASFP can work in any extensible store, like Redis [43]. Splinter's kernel-bypass networking simplifies cost modeling; modeling kernel TCP costs would add complexity but would increase potential savings over our implementation. ASFP is also targeted toward diverse, multi-tenant workloads with heterogeneous operations that it can place. Single-application functions added to a single-tenant store could likely be statically classified as server- or client-side, eliminating some benefits of dynamic profiling.

Network congestion. Congestion isn't a problem in our high-bandwidth setup. However, exposing transport layer information (window sizes) to ASFP could let it choose placement to minimize network traffic under congestion. If an invocation accesses little data and enqueues many bytes for transmission while the network is congested, the server could return the data instead, forcing the client to compute the result.

6 Related Work

Adaptive pushback for Splinter builds on many ideas.

Storage Procedures, UDFs, and Database Extensions. There are several common approaches for pushing computation to databases and data stores. SQL is ubiquitous, though it is a poor fit for specialized computation, especially for microsecond timescales. SQL stored procedures [34] and UDFs [19, 22, 33, 38, 48] allow more specialized, procedural logic to be added to stores, and they can often be compiled for performance. Some databases also allow dynamic libraries to be loaded as well for specialized operations [49]. Some key-value stores and object stores support similar user-provided functions or extensions provided either at server-start time [43] or at runtime [14, 26, 45, 53], some relying on just-in-time compilation and some ahead-of-time compiled.

All of these approaches can ship computation to storage, but they do not address the question of whether doing so is beneficial for storage servers or its clients. Our approach could be applied to stored procedures and UDFs.

Thread and Process Migration. In the 1990s, both process and thread migration were pursued as ways to move computation at a fine-grain, often to place computation near data [10, 32, 35]. These approaches are often complex and highly runtime-specific, since moving in-progress computation requires precise reasoning about the state it closes over. We take a much simpler approach; rather than moving running functions, we preserve some of the work they have done through their read/write sets and restart functions from the beginning at clients. This assumes that invocations tend to be short, which is true for the small timescales that we target.

Fast, Disaggregated Storage. Fast networks have led to disaggregated storage and even disaggregated memory [16, 30, 31]. Many works focus on building scalable in-memory stores that move data efficiently [29], and many more have used techniques like kernel-bypass and RDMA (both one-sided and two-sided) to minimize the CPU cost of request processing for fast storage [20, 21, 29, 52]. These approaches reduce server-side CPU consumption and improve throughput for the simple operations that these stores provide, but they provide no way to move computation into storage when it would improve server efficiency. FaRM [12] is an exception. Clients can do this manually since each node in FaRM is both a client and a server; functions can be compiled into the storage server for custom request handlers. However, FaRM lacks an adaptive mechanism to move invocations of these functions between clients and remote servers.

Cell [36] is a distributed in-memory B-tree that uses RDMA. Cell uses a similar idea to pushback. In Cell, when clients lookup keys in the B-tree they can use one-sided RDMA reads to fetch nodes from the B-tree and perform the tree traversal client-side, or clients can send a request to a server to have it do the traversal. Clients track round-trip times to estimate queuing delay to determine whether the server network card or server CPU is under pressure. This lets them intelligently choose between the two approaches to improve server throughput. Our approach is similar, but ASFP is black box; it assumes no visibility into the functions that clients want to run. As a result, it must track and predict the relative client-side versus server-side cost of operations.

Offloading and migrating code has also been pursued in other contexts like edge computing and mobile devices where there is a large imbalance between the capabilities of devices and where moving data over edge links incurs high cost [8,15, 18,24,46]. Our approach and Splinter are also similar to Active Disks [1, 7] that allow application code to be downloaded to and executed on disk- and flash-based storage devices.

7 Conclusion

Today, data center and cloud storage systems disaggregate compute; clients must fetch data to compute on it, resulting in wasted work. When clients can send computation to storage, both clients and storage servers can benefit; however, to be practical, storage servers need a means to avoid becoming a bottleneck. ASFP does this by keeping client functions logically decoupled from storage and deciding physical placement of their invocations at runtime. By profiling invocations and observing both the CPU costs and savings they create at the server, storage servers can dynamically determine when invocations should be forced back for client-side execution.

We show ASFP's promise; servers and smart clients adapt function placement at microsecond timescales, improving throughput even when storage function CPU cost varies. We show it works when running a mix of different applications' logic, providing better throughput than running that logic purely at storage servers (85% more) or clients (10% more).

Acknowledgments. Thanks to Mazhar Naqvi and Jacob Barzee for contributing to ASFP, to the reviewers for their comments, and to our shepherd, Changwoo Min. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Facebook and VMware.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 81–91, New York, NY, USA, 1998. ACM.

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast Key-value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 113–119. ACM, 2019.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 23–34, 1995.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 249–266, USA, 2019. USENIX Association.

[7] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[8] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[9] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+ T: Benchmarking Web-scale Transactional Databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[10] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[11] DPDK Project. Data Plane Development Kit. http://dpdk.org/. Accessed: 2020-01-15.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[14] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An Active Distributed Key-value Store. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 323–336, 2010.

[15] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code Offload by Migrating Execution Transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, Hollywood, CA, 2012. USENIX.

[16] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[17] Greg Hamerly, Charles Elkan, et al. Bayesian Approaches to Failure Prediction for Disk Drives. In *ICML*, volume 1, pages 202–209, 2001.

[18] Mor Harchol-Balter and Allen B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, August 1997.

[19] Guy Harrison and Steven Feuerstein. *MySQL stored procedure programming*. " O'Reilly Media, Inc.", 2006.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, 2016. USENIX Association.

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.

[23] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[24] Michael Kozuch and Mahadev Satyanarayanan. Internet suspend/resume. In *WMCSA*, volume 2, page 40, 2002.

[25] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405. ACM, 2017.

[26] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, 2018. USENIX Association.

[27] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[28] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard Drive Failure Prediction Using Classification and Regression Trees. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 383–394. IEEE, 2014.

[29] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[30] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ACM SIGARCH computer architecture news*, volume 37, pages 267–278. ACM, 2009.

[31] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level Implications of Disaggregated Memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[32] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1997.

[33] Microsoft, Inc. Stored Procedures (Database Engine) - SQL Server. `https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-2017`. Accessed: 2020-01-15.

[34] Microsoft, Inc. User-Defined Functions - SQL Server. `https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-2017`. Accessed: 2020-01-15.

[35] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000.

[36] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, 2016. USENIX Association.

[37] JF Murray, GF Hughes, and K Kreutz-Delgado. Comparison of Machine Learning Methods for Predicting Failures in Hard Drives. *Journal of Machine Learning Research*, 6, 2005.

[38] Oracle, Inc. Oracle PL/SQL. http://www.oracle.com/technetwork/database/features/plsql/index.html. Accessed: 2020-01-15.

[39] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[40] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4):92–105, December 2009.

[41] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[42] Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[43] Redis. http://redis.io/. Accessed: 2020-01-15.

[44] Felix Salfner, Maren Lenk, and Miroslaw Malek. A Survey of Online Failure Prediction Methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[45] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the 12th European Conference on Computer Systems*, Eurosys '17, pages 175–190. ACM, 2017.

[46] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[47] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.
.

[48] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[49] The PostgreSQL Global Development Group. PostgreSQL: Documentation: 10: H.4. Extensions. http://www.postgresql.org/docs/10/static/external-extensions.html. Accessed: 2020-01-15.

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[51] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[52] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[53] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

NetKernel: Making Network Stack Part of the Virtualized Infrastructure

Zhixiong Niu
Microsoft Research

Hong Xu
City University of Hong Kong

Peng Cheng
Microsoft Research

Qiang Su
City University of Hong Kong

Yongqiang Xiong
Microsoft Research

Tao Wang
New York University

Dongsu Han
KAIST

Keith Winstein
Stanford University

Abstract

This paper presents a system called NetKernel that decouples the network stack from the guest virtual machine and offers it as an independent module. NetKernel represents a new paradigm where network stack can be managed as part of the virtualized infrastructure. It provides important efficiency benefits: By gaining control and visibility of the network stack, operator can perform network management more directly and flexibly, such as multiplexing VMs running different applications to the same network stack module to save CPU. Users also benefit from the simplified stack deployment and better performance. For example mTCP can be deployed without API change to support nginx natively, and shared memory networking can be readily enabled to improve performance of colocated VMs. Testbed evaluation using 100G NICs shows that NetKernel preserves the performance and scalability of both kernel and userspace network stacks, and provides the same isolation as the current architecture.

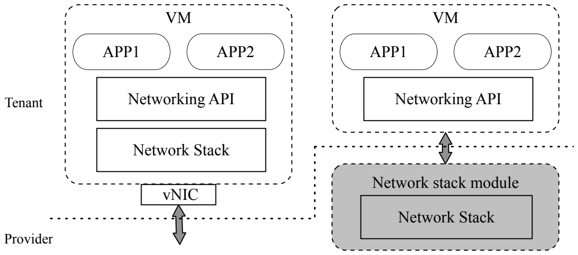
1 Introduction

Virtual machine (VM) is the predominant virtualization form in today's cloud due to its strong isolation guarantees. VMs allow customers to run applications in a wide variety of operating systems (OSes) and configurations. VMs are also heavily used by cloud operators to deploy internal services, such as load balancing, proxy, VPN, etc., both in a public cloud for tenants and in a private cloud for various business units of an organization.

VM based virtualization largely follows traditional OS design. In particular, the TCP/IP network stack is encapsulated inside the VM as part of the guest OS as shown in Figure 1(a). Applications own the network stack, which is separated from the network infrastructure that operators own; they interface using the virtual NIC abstraction. This architecture preserves the familiar hardware and OS abstractions so a vast array of workloads can be easily moved into the cloud. It also provides high flexibility to applications to customize the entire network stack.

We argue that the current division of labor between application and network infrastructure is becoming increasingly inadequate, especially in a private cloud setting. The central issue is that the network stack is controlled solely by individual guest VM; the operator has almost zero visibility or control. This leads to efficiency problems that manifest in various aspects of running the cloud network. Firstly, the operator is unable to orchestrate resource allocation at the end-points of the network fabric, resulting in low resource utilization. It remains difficult today for the operator to meet or define performance SLAs despite much prior work [17,28,35,41,56,57], as she cannot precisely provision resources just for the network stack or control how the stack consumes these resources. Further, resources (e.g. CPU) have to be provisioned on a per-VM basis based on the peak traffic; it is impossible to coordinate across VM boundaries. This degrades the overall utilization of the network stack since in practice traffic to individual VMs is extremely bursty. Also, many network management tasks like monitoring, diagnosis, and troubleshooting have to be performed in an extra layer outside the guest VMs, which requires significant efforts in design and implementation [23, 59, 60]. They can be done more efficiently if the network stack is opened up to the operator.

Even the simple task of maintaining or deploying a network stack suffers from much inefficiency today. Numerous new stack designs and optimizations ranging from congestion control [14, 19, 50], scalability [34, 42], zerocopy datapath [4, 34, 55, 64, 65], NIC multiqueue scheduling [63], etc. have been proposed in our community. Yet the operator, with sufficient expertise and resources, could not easily deploy these solutions in a virtualized cloud to improve performance and reduce overheads because it does not own or control the network stack. As a result, our community is still finding ways to deploy DCTCP in the public cloud [20, 31, 36]. On the other hand, applications without much knowledge of the underlying network or expertise on networking are forced to juggle the deployment and maintenance details. For example if one wants to deploy a new stack like mTCP [34], he faces a host of problems such as setting up kernel bypass, testing



(a). Existing architecture (b). Decoupling network stack from the guest

Figure 1: Decoupling network stack from the guest, and making it part of the virtualized infrastructure.

with kernel versions and NIC drivers, and porting applications to the new APIs. Given the intricacy of implementation and the velocity of development, it is a daunting task for individual users, whether tenants in a public cloud or first-party services in a private cloud, to maintain the network stack all by themselves.

To address these limitations, we advocate the *separation* of network stack from the guest OS as a new paradigm, in which the network stack is managed as part of the virtualized infrastructure by the operator. As the heavy-lifting is taken care of, applications can just use network stack as a basic service of the infrastructure and focus on their business logic.

More concretely, as shown in Figure 1(b), we propose to decouple the VM network stack from the guest OS. We keep the network APIs such as BSD sockets intact, and use them (instead of vNIC) as the abstraction boundary between application and infrastructure. Each VM is served by an external network stack module (NSM) that runs the network stack chosen by the user , *e.g.,* the kernel-bypass stack mTCP or the improved kernel stack FastSocket [42] . Application data are handled in the NSM, whose design and implementation are managed by the operator. Various network stacks can be provided as different NSMs to ensure applications with diverse requirements can work. This new paradigm does not necessarily enforce a single transport design, or trade off such flexibility of the existing architecture.

We make three specific contributions in this paper.

- We design and implement NetKernel that demonstrates our new approach is feasible on existing KVM virtualization platforms (§3–§5). NetKernel provides transparent BSD socket redirection so existing applications can run directly.
- We present NetKernel's benefits by showcasing novel use cases that are difficult to realize today (§6). For example, we show that NetKernel enables multiplexing: one NSM can serve multiple VMs at the same time and save over 40% CPU cores without degrading performance using traces from a production cloud.
- We conduct comprehensive testbed evaluation with commodity 100G NICs to show that NetKernel achieves the same scalability and isolation as the current architecture (§7). For example, the kernel stack NSM achieves 100G send throughput with 3 cores; the mTCP NSM achieves 979Krps with 8 cores.

NetKernel's official website is https://netkernel.net.

2 Motivation

Decoupling the network stack from the guest OS and making it part of the infrastructure marks a clear departure from the way networking is provided to VMs nowadays. In this section we elaborate why this is a better architectural design by presenting its benefits and tradeoffs, and contrasting it with alternative solutions.

2.1 Benefits and Tradeoffs

We highlight the key benefits of our vision with several new use cases that we experimentally realize with NetKernel in §6.

Better efficiency in management for the operator. Gaining control over the network stack, the operator can now perform network management more efficiently. For example it can orchestrate the resource provisioning strategies more flexibly: For mission-critical workloads, it can dedicate CPU resources to their NSMs to offer performance SLAs in terms of throughput and rps (requests per second) guarantees. For elastic workloads, on the other hand, it can consolidate their VMs to the same NSM (if they use the same network stack) to improve its resource utilization. The operator can also directly implement management functions as an integral part of user's network stack, compared to doing them in an extra layer outside the guest OS.

Use case 1: Multiplexing (§6.1). Utilization of network stack in VMs is very low most of the time in practice. Using a real trace from a large cloud, we show that NetKernel enables multiple VMs to be multiplexed onto one NSM to serve the aggregated traffic and saves over 40% CPU cores for the operator without performance degradation.

Deployment and performance gains for users. Making network stack part of the virtualized infrastructure is also beneficial for users in both public and private clouds. Various kernel stack optimizations [42, 64], high-performance userspace stacks [11, 18, 34, 55], and even designs using advanced hardware [6, 8, 9, 43] can now be deployed and maintained transparently without user involvement or application code change. For instance, DCTCP can now be deployed across the board easily in a public cloud. Since the BSD socket is the only abstraction exposed to the applications, it is now feasible to adopt new stack designs independent of the guest kernel or the network API. Our vision also opens up new design space by allowing the network stack to exploit visibility of the infrastructure for performance benefits.

Use case 2: Deploying mTCP without API change (§6.2). We show that NetKernel enables unmodified applications in the VM to use mTCP [34] in the NSM, and improves

Paradigm	Scenario	Multiplexing	New Stack Deployment	Performance Opt. with Infrastructure
Guest-based	VM	✗	Require user effort	✗
Host-based	Container	✓	Limited by host OS	✓
Application-based	Library OS	✗	Require user effort	✗
NetKernel	VM + NSM	✓	✓	✓

Table 1: Comparison of different network stack architectures depending on where the stack is. The current architecture is a guest-based paradigm where the network stack is part of the guest OS of a VM.

performance greatly due to mTCP's kernel bypass design. mTCP is a userspace stack with new APIs (including modified `epoll`/`kqueue`). During the process, we also find and fix a compatibility issue between mTCP and our NIC driver, and save significant maintenance time and effort for users.

Use case 3: Shared memory networking (§6.3). When two VMs of the same user are colocated on the same host, NetKernel can directly detect this and copy their data via shared memory to bypass TCP stack processing and improve throughput. This is difficult to achieve today as VMs have no knowledge about the underlying infrastructure [40, 66].

Tradeoffs. We are conscientious of the tradeoffs our approach brings about. For example, due to the removal of vNIC and redirection from the VM's own network stack, some networking tools like netfilter are affected. This is acceptable since most users wish to focus on their applications instead of tuning a network stack. If they wish to gain maximum control over the network stack they can still use VMs without NetKernel. Also, additional fate-sharing may be introduced by our approach say when multiple VMs share the same NSM. We believe this is not serious because cloud users already have fate-sharing with the vSwitch, hypervisor, and the complete virtual infrastructure. The efficiency benefits of our approach as demonstrated outweigh the marginal increase of fate-sharing; the success of cloud computing these years is another strong testament to this tradeoff. NetKernel enforces another level of indirection in order to achieve flexibility which does not cause performance degradation in most cases as we will show in §7, and part of it can run on hardware for more efficiency (see §8). Lastly, one may have security concerns with using the NSM to handle tenant traffic. Most of the security protocols such as HTTPS/TLS work at the application layer and are not affected. One exception is IPSec. Due to the certificate exchange issue, IPSec does not work in our approach. However, in practice IPSec is implemented at dedicated gateways instead of end-hosts [62]. Thus we believe the impact is not serious. More discussion on security can be found in §8.

2.2 Alternative Solutions

We now discuss several alternative architectures depending on where the network stack resides, and why they are inadequate compared to NetKernel as summarized in Table 1. Note that none of them provides all four key benefits as NetKernel does.

Host-based. The first alternative is a host-based paradigm where the network stack runs on the host machine. This corresponds to the container scenario in the cloud. A container is essentially a process with namespace isolation: it shares the host's network stack in the hypervisor. Therefore containers can achieve some of NetKernel's benefits, i.e multiplexing and performance optimization with infrastructure, since the operator can access the hypervisor. However, container has tight coupling with the host OS which makes the stack deployment difficult. A Windows application in a container cannot use the Linux-based mTCP, unless the operator ports mTCP to Windows. With NetKernel no such porting is needed: mTCP can run in a Linux-based NSM and serve a Windows user because the only coupling is the BSD socket APIs.

We also note that currently containers have performance isolation problems [38] and as a result are usually constrained to be deployed inside VMs in production settings. In fact we find that all major public clouds [1, 2, 5] require users to launch containers inside VMs. Thus, our work is centered around VMs that cover the majority of usage scenarios in a cloud. NetKernel readily benefits containers running inside VMs as well.

Application-based. Another alternative is to move the network stack upwards by taking an application-based paradigm. A representative scenario is library OS including unikernels [22, 44] and microkernels [26], where many OS services including the network stack are packaged as libraries and compiled with the application in userspace. Similar to the guest-based paradigm, users have to deploy the network stack by themselves though the I/O performance can be improved with unikernels [46] and microkernels. In addition, application-based paradigm is a clean-slate approach and requires radical changes to both the virtualization software and user applications. NetKernel can flexibly decouple the network stack from the guest without re-writing existing applications or hypervisor.

3 Design Philosophy

NetKernel imposes three fundamental design questions around the separation of network stack from the guest OS:

1. How to transparently redirect socket API calls without changing applications?
2. How to transmit the socket semantics between the VM and NSM whose implementation of the stack may vary?

3. How to ensure high performance with semantics transmission (e.g., 100 Gbps)?

These questions touch upon a largely uncharted territory in the design space. Thus our main objective in this paper is to demonstrate feasibility of our approach on existing virtualization platforms and showcase its potential. Performance and overhead are not our primary goals. It is also not our goal to improve any particular network stack design.

In answering the questions above, NetKernel's design has the following highlights.

Transparent socket API redirection. NetKernel needs to redirect BSD socket calls to the NSM instead of the tenant network stack. This is done by inserting into the guest a library called GuestLib. The GuestLib provides a new socket type called NetKernel socket with a complete implementation of BSD socket APIs. It replaces all TCP and UDP sockets when they are created with NetKernel sockets, effectively redirecting them without changing applications.

A lightweight semantics channel. Different network stacks may run as different NSMs, so NetKernel needs to ensure socket semantics from the VM work properly with the actual NSM stack implementation. For this purpose NetKernel builds a lightweight socket semantics channel between VM and its NSM. The channel relies on small fix-sized queue elements as intermediate representations of socket semantics: each socket API call in the VM is encapsulated into a queue element and sent to the NSM, who would effectively translate the queue element into the corresponding API call of its network stack.

Scalable lockless queues. As NIC speed in cloud evolves from 40G/50G to 100G [24] and higher, the NSM has to use multiple cores for the network stack to achieve line rate. NetKernel thus adopts scalable lockless queues to ensure VM-NSM socket semantics transmission is not a bottleneck. Each core services a dedicated set of queues so performance is scalable with number of cores. More importantly, each queue is memory shared with a software switch, so it can be lockless with only a single producer and a single consumer to avoid expensive lock contention [33, 34, 42].

Switching the queue elements offers important benefits beyond lockless queues. It facilitates a flexible mapping between VM and NSM: a NSM can support multiple VMs without adding more queues compared to binding the queues directly between VM and NSM. In addition, it allows dynamic resource management: cores can be readily added to or removed from a NSM, and a user can switch her NSM on the fly. The CPU overhead of software switching can be addressed by hardware offloading [24, 27], which we discuss in §7.4 in more detail.

VM based NSM. Lastly we discuss an important design choice regarding the NSM. The NSM can take various forms. It may be a full-fledged VM with a monolithic kernel. Or it can be a container or module running on the hypervisor, which is appealing because it consumes less resource and offers better performance. Yet it entails porting a complete TCP/IP stack to the hypervisor. Achieving memory isolation among containers or modules are also difficult [52]. More importantly, it introduces another coupling between the network stack and the hypervisor, which defeats the purpose of NetKernel. Thus we choose to use a VM for NSM. VM based NSM readily supports existing kernel and userspace stacks from various OSes. VMs also provide good isolation and we can dedicate resources to a NSM to guarantee performance. VM based NSM is the most flexible: we can run stacks independent of the hypervisor.

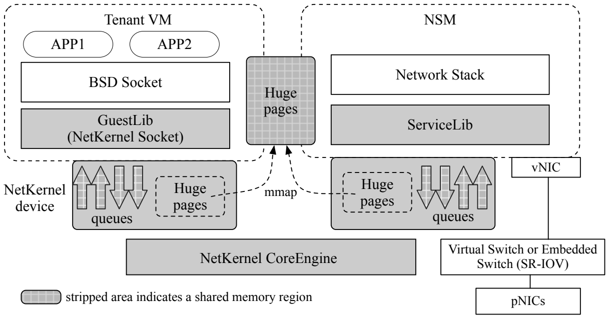


Figure 2: NetKernel design overview.

4 Design

Figure 2 depicts NetKernel's architecture. The BSD socket APIs are transparently redirected to a complete NetKernel socket implementation in GuestLib in the guest kernel (§4.1). The GuestLib can be deployed as a kernel patch and is the only change we make to the user VM. Network stacks are implemented by the operator on the same host as Network Stack Modules (NSMs), which are individual VMs in our current design. Inside the NSM, a ServiceLib interfaces with the network stack. The NSM connects to the vSwitch, be it a software or a hardware switch, and then the pNICs. Thus our design also supports SR-IOV.

All socket operations and their results are translated into NetKernel Queue Elements (NQEs) by GuestLib and ServiceLib (§4.2). For NQE transmission, GuestLib and ServiceLib each has a NetKernel device, or NK device in the following, consisting of one or more sets of lockless queues. Each queue set has a *send queue* and *receive queue* for operations with data transfer (e.g. `send()`), and a *job queue* and *completion queue* for control operations without data transfer (e.g. `setsockopt()`). Each NK device connects to a software switch called CoreEngine, which runs on the hypervisor and performs actual NQE switching (§4.3). The CoreEngine is also responsible for various management tasks such as setting up the NK devices, ensuring isolation among VMs, etc. (§4.4) A unique set of hugepages are shared between each VM-NSM tuple for application data exchange. A NK device also maintains a hugepage region that is memory mapped to the corresponding application hugepages as in Figure 2

(§4.5). Note that as the socket API that copies data is preserved, misbehaving applications cannot pose security risks on NetKernel, this is the same as original kernel design. We discuss additional security implications of NetKernel in §8.

For ease of presentation, we assume both the user VM and NSM run Linux, and the NSM uses the kernel stack.

4.1 Transparent Socket API Redirection

We first describe how NetKernel's GuestLib interacts with applications to support BSD socket semantics transparently.

Kernel space API redirection. There are essentially two approaches to redirect BSD socket calls to NSM, each with its unique tradeoffs. One is to implement it in userspace using `LD_PRELOAD` for example. The advantages are: (1) It is efficient without syscall overheads and performance is high [34]; (2) It is easy to deploy without kernel modification. However, this implies each application needs to have its own redirection service, which limits the usage scenarios. Another way is kernel space redirection, which naturally supports multiple applications without IPC. The flip side is that performance may be lower due to context switching and syscall overheads.

We opt for kernel space API redirection to support most of the usage scenarios, and leave userspace redirection as future work. GuestLib is a kernel module deployed in the guest. This is feasible by distributing images of para-virtualizated guest kernels to users, a practice operators are already doing nowadays. Note that kernel space redirection follows the asynchronous syscall model [61] to get better performance.

NetKernel socket API. GuestLib creates a new type of sockets—`SOCK_NETKERNEL`, in addition to TCP (`SOCK_STREAM`) and UDP (`SOCK_DGRAM`) sockets. It registers a complete implementation of BSD socket APIs to the guest kernel. When the guest kernel receives a `socket()` call to create a new TCP socket say, it replaces the socket type with `SOCK_NETKERNEL`, creates a new NetKernel socket, and initializes the socket data structure with function pointers to NetKernel socket implementation in GuestLib. The `sendmsg()` for example now points to `nk_sendmsg()` in GuestLib instead of `tcp_sendmsg()`.

4.2 A Lightweight Semantics Channel

Socket semantics are contained in NQEs and carried around between GuestLib and ServiceLib via their respective NK devices.

1B	1B	1B	4B	8B	8B	4B	5B
op type	VM ID	Queue set ID	VM socket ID	op_data	data pointer	size	rsved

Figure 3: Structure of a NQE. Here socket ID denotes a pointer to the `sock` struct in the user VM or NSM, and is used for NQE transmission with VM ID and queue set ID in §4.3; `op_data` contains data necessary for socket operations, such as ip address for bind; `data pointer` is a pointer to application data in hugepages; and `size` is the size of pointed data in hugepages.

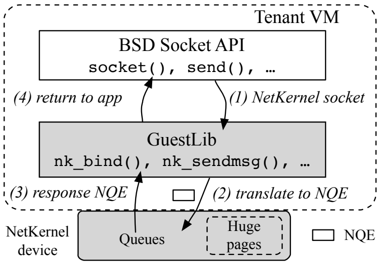


Figure 4: NetKernel socket implementation in GuestLib redirects socket API calls. GuestLib translates socket API calls to NQEs and ServiceLib translates results into NQEs as well (not shown here).

NQE and socket semantics translation. Figure 3 shows the structure of a NQE with a fixed size of 32 bytes. Translation happens at both ends of the semantics channel: GuestLib encapsulates the socket semantics into NQEs and sends to ServiceLib, which then invokes the corresponding API of its network stack to execute the operation; the execution result is again turned into a NQE in ServiceLib first, and then translated by GuestLib back into the corresponding response of socket APIs.

For example in Figure 4, to handle the `socket()` call in the VM, GuestLib creates a new NQE with the operation type and information such as its VM ID for NQE transmission. The NQE is transmitted by GuestLib's NK device. The `socket()` call now blocks until a response NQE is received. After receiving the NQE, ServiceLib parses the NQE from its NK device, invokes the `socket()` of the kernel stack to create a new TCP socket, prepares a new NQE with the execution result, and enqueues it to the NK device. GuestLib then receives and parses the response NQE and wakes up the `socket()` call. The `socket()` call now returns to application with the NetKernel socket file descriptor (fd) if a TCP socket is created at the NSM, or with an error number consistent with the execution result of the NSM.

We defer the handling of application data to §4.5.

Queues for NQE transmission. NQEs are transmitted via one or more sets of queues in the NK devices. A queue set has four independent queues: a *job queue* for NQEs representing socket operations issued by the VM without data transfer, a *completion queue* for NQEs with execution results of control operations from the NSM, a *send queue* for NQEs representing operations issued by VM with data transfer; and a *receive queue* for NQEs representing events of newly received data from NSM. Queues of different NK devices have strict correspondence: the NQE for `socket()` for example is put in the job queue of GuestLib's NK device, and sent to the job queue of ServiceLib's NK device.

We now present the working of I/O event notification mechanisms like epoll with the receive queue. Figure 5 depicts the details. Suppose an application issues `epoll_wait()` to monitor some sockets. Since all sockets are now NetKernel sockets, the `nk_poll()` is invoked by `epoll_wait()` and checks the receive queue to see if there is any NQE

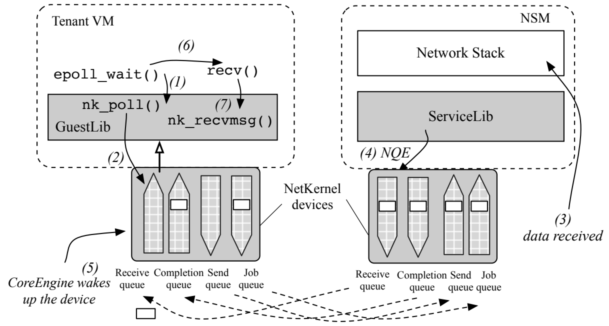


Figure 5: The socket semantics channel with epoll as an example. GuestLib and ServiceLib translate semantics to NQEs, and queues in the NK devices perform NQE transmission. Job and completion queues are for socket operations and execution results, send queues are for socket operations with data, and receive queues are for events of newly received data. Application data processing is not shown.

for this socket. If yes, this means there are new data received, `epoll_wait()` then returns and the application issues a `recv()` call with the NetKernel socket fd of the event. This points to `nk_recvmsg()` which parses the NQE from receive queue for the data pointer, copies data from the hugepage directly to the userspace, and returns.

If `nk_poll()` does not find any relevant NQE, it sleeps until CoreEngine wakes up the NK device when new NQEs arrive to its receive queue. GuestLib then parses the NQEs to check if any sockets are in the epoll instances, and wakes up the epoll to return to application. An `epoll_wait()`can also be returned by a timeout.

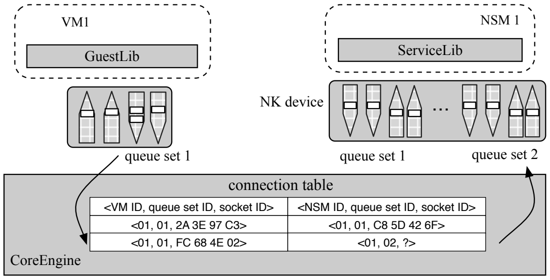
4.3 NQE Switching across Lockless Queues

We now elaborate how NQEs are switched by CoreEngine and how the NK devices interact with CoreEngine.

Scalable queue design. The queues in a NK device is scalable: there are one dedicated queue set per vCPU for both VM and NSM, so NetKernel performance scales with CPU resources. Each queue set is shared memory with the CoreEngine, essentially making it a single producer single consumer queue without lock contention. VM and NSM may have different numbers of queue sets.

Switching NQEs in CoreEngine. NQEs are load balanced across multiple queue sets with the CoreEngine acting as a switch. CoreEngine maintains a connection table as shown in Figure 6, which maps the tuple ⟨VM ID, queue set ID, socket ID⟩ to the corresponding ⟨NSM ID, queue set ID, socket ID⟩ and vice versa. Here a socket ID corresponds to a pointer to the `sock` struct in the user VM or NSM. We call them VM tuple and NSM tuple respectively. NQEs only contain VM tuple information.

Using the running example of the `socket()` call, we can see how CoreEngine uses the connection table. The process is also shown in Figure 6. (1) When CoreEngine processes the socket NQE from VM1's queue set 1, it realizes this is a new connection, and inserts a new entry to the table with the VM tuple from the NQE. (2) It checks which NSM should handle it,[1] performs hashing based on the three tuple to determine which queue set (say 2) to switch to if there are multiple queue sets, and copies the NQE to the NSM's corresponding job queue. CoreEngine adds the NSM ID and queue set ID to the new entry. (3) ServiceLib gets the NQE and copies the VM tuple to its response NQE, and adds the newly created connection ID in the NSM to the `op_data` field of response NQE. (4) CoreEngine parses the response NQE, matches the VM tuple to the entry and adds the NSM socket ID to complete it, and copies the response NQE to the completion queue 1 of VM1 as instructed in the NQE. Later NQEs for this VM connection can be processed by the correct NSM connection and vice versa. ServiceLib pins its connections to its vCPUs and queue sets, so processing the NQE and sending the response NQE are done on the same CPU.



<VM ID, queue set ID, socket ID>	<NSM ID, queue set ID, socket ID>
<01, 01, 2A 3E 97 C3>	<01, 01, C8 5D 42 6F>
<01, 01, FC 68 4E 02>	<01, 02, ?>

Figure 6: NQE switching with CoreEngine.

The connection table allows flexible multiplexing and demultiplexing with the socket ID information. For example one NSM can serve multiple VMs using different sockets. CoreEngine polls all queue sets to maximize performance.

4.4 Management with CoreEngine

CoreEngine acts as the control plane of NetKernel and carries out many control tasks beyond NQE switching.

NK device and queue setup. CoreEngine allocates shared memory for the queue sets and sets up the NK devices accordingly when a VM or NSM starts up, and de-allocates when they shut down. Queues can also be dynamically added or removed with the number of vCPUs.

Isolation. CoreEngine sits in an ideal position to carry out isolation among VMs. In our design CoreEngine polls each queue set in a round-robin fashion to ensure the basic fair sharing. Operator can implement other isolation mechanisms to rate limit a VM in terms of bandwidth or the number of NQEs (i.e. operations) per second, which we show in §7.3. Note that CoreEngine isolation happens for egress; ingress isolation at the NSM is more challenging and may have to use physical NIC queues [21].

Busy-polling. The busy-polling design of CoreEngine requires a dedicated core per machine which is an inherent

[1] A user VM to NSM mapping is determined either by the users/operator offline or some load balancing scheme dynamically by CoreEngine.

overhead of our design. We resort to this simple design as we focus on showing feasibility and potential of NetKernel in this work, and prior work also used dedicated cores for software polling [40]. One can explore hardware offloading using FPGAs for example to eliminate this overhead [23, 24].

4.5 Processing Application Data

We now discuss the last missing piece of NetKernel design: how application data are actually processed in the system.

Sending data. Data is transmitted by hugepages shared between the VM and NSM. Their NK devices maintain a hugepage region that is mmaped to the application hugepages. For sending data with `send()`, GuestLib copies data from userspace directly to the hugepage, and adds a data pointer to the send NQE. It also increases the send buffer usage for this socket similar to the send buffer size maintained by the kernel. The `send()` now returns to application. ServiceLib invokes `tcp_sendmsg()` provided by the kernel stack upon receiving the send NQE. Data are obtained from hugepages, processed by the network stack, and sent to the vNIC. A new NQE is generated with the result of send by the NSM and sent to GuestLib, who then decreases the send buffer usage.

Receiving data. Now for receiving packets in the NSM, a normal network stack would send received data to userspace applications. In order to send received data to the user VM, ServiceLib then copies the data chunk to hugepages and create a new NQE to the receive queue, which is then sent to the VM. It also increases the receive buffer usage for this connection, similar to the send buffer maintained by GuestLib described above. The rest of the receive process is already explained in §4.2. Note that application uses `recv()` to copy data from hugepages to their own buffer.

ServiceLib. As discussed ServiceLib deals with much of data processing at the NSM side so the network stack works in concert with the rest of NetKernel. One thing to note is that unlike the kernel space GuestLib, ServiceLib should live in the same space as the network stack to ensure best performance. We have focused on a Linux kernel stack with a kernel space ServiceLib here. The design of a userspace ServiceLib for a userspace stack is similar in principle. ServiceLib busy-polls its queues for maximum performance.

4.6 Optimization

We present several optimizations employed in NetKernel.

Pipelining. NetKernel applies pipelining between VM and NSM for performance. For example on the VM side, a `send()` returns immediately after putting data to the hugepages, instead of waiting for the actual send result from the NSM. Similarly the NSM would handle `accept()` by accepting a new connection and returning immediately, before the corresponding NQE is sent to GuestLib and then application to process. Doing so does not break BSD socket semantics. Take `send()` for example. A successful `send()` does not guarantee delivery of the message [13]; it merely indicates the message is written to socket buffer successfully. In NetKernel a successful `send()` indicates the message is written to buffer in the hugepages successfully. As explained in §4.5 the NSM sends the result of send back to the VM to indicate if the socket buffer usage can be decreased or not.

Interrupt-driven polling. We adopt an interrupt-driven polling design for NQE event notification to GuestLib's NK device. This is to reduce the overhead of GuestLib and user VM. When an application is waiting for events e.g. the result of the `socket()` call or receive data for epoll, the device will first poll its completion queue and receive queue. If no new NQE comes after a short time period (20µs in our experiments), the device sends an interrupt to CoreEngine, notifying that it is expecting NQE, and stops polling. CoreEngine later wakes up the device, which goes back to polling mode to process new NQEs from the completion queue. This is similar in spirit to busy-polling sockets in Linux kernel [3, 10]. Interrupt-driven polling presents a favorable trade-off between overhead and performance compared to pure polling based or interrupt based design. It saves precious CPU cycles when load is low and ensures the overhead of NetKernel is very small to the user VM. Performance on the other hand is competent since the response NQE is received within the polling period in most cases for blocking calls, and when the load is high polling automatically drives the notification mechanism. As explained before CoreEngine and ServiceLib use busy polling to maximize performance.

Batching. Batching is used in many parts of NetKernel for better throughput. CoreEngine uses batching whenever possible for polling from and copying into the queues. The NK devices also receive NQEs in a batch.

5 Implementation

Our implementation is based on QEMU KVM 2.5.0 and Linux kernel 4.9 for both host and guest, with over 11K LoC.

GuestLib. We add the `SOCK_NETKERNEL` socket to the kernel (`net.h`), and modify `socket.c` to rewrite the `SOCK_STREAM` to `SOCK_NETKERNEL` during socket creation. We implement GuestLib as a kernel module with two components: Guestlib_core and nk_driver. Guestlib_core is mainly for Netkernel sockets and NQE translation, and nk_driver is for NQE communications via queues. Guestlib_core and nk_driver communicate with each other using function calls.

ServiceLib and NSM. We also implement ServiceLib as two components: Servicelib_core and nk_driver. Servicelib_core translates NQEs to network stack APIs, and the nk_driver is identical to the one in GuestLib. For the kernel stack NSM, Servicelib_core calls the kernel APIs directly to handle socket operations without entering userspace. We create an independent `kthread` to poll the job queue and send queue for NQEs to avoid kernel stuck. Some BSD socket APIs can not be invoked in kernel space directly. We use `EXPORT_SYMBOLS` to export the functions for ServiceLib. Meanwhile, the boundary check between kernel space and userspace is disabled. We use per-core `epoll_wait()` to obtain incoming events from the kernel stack.

We also port mTCP [12] as a userspace stack NSM. It uses DPDK 17.08 for packet I/O. For simplicity, we maintain its two-thread model and per-core data structure. We implement the NSM in mTCP's application thread at each core. The ServiceLib is essentially an mTCP application: once receiving a NQE from its send queue, it accesses data from the shared hugepage by the data pointer in the NQE and sends it using mTCP with DPDK. For receiving, the received data is copied into the hugepage, and ServiceLib encapsulates the data pointer into a NQE of the receive queue. The per-core application thread (1) translates NQEs polled from the NK device to mTCP socket APIs, and (2) responds NQEs to the tenant VM based on the network events collected by `mtcp_epoll_wait()`. Since mTCP works in non-blocking mode for performance, we buffer send operations at each core and set the `timeout` parameter to 1ms in `mtcp_epoll_wait()` to avoid starvation when polling NQE requests.

Queues and hugepages. The hugepages are implemented based on QEMU's IVSHMEM. The page size is 2 MB and we use 128 pages. The queues are ring buffers implemented as much smaller IVSHMEM devices. Together they form a NK device which is a virtual device to the VM and NSM.

CoreEngine. The CoreEngine is a daemon with two threads on the KVM hypervisor. One thread listens on a pre-defined port to handle NK device (de)allocation requests, namely 8-byte network messages of the tuples ⟨ce_op, ce_data⟩. When a VM (or NSM) starts (or terminates), it sends a request to CoreEngine for registering (or deregistering) a NK device. If the request is successfully handled, CoreEngine responds in the same message format. Otherwise, an error code is returned. The other thread polls NQEs in batches from all NK devices and switches them as described in §4.3.

6 Evaluation: New Use Cases

In the first part of evaluation, we present some new use cases that are realized using our prototype to demonstrate the potential of NetKernel. Details of the performance and overhead microbenchmarks are presented in §7.

6.1 Multiplexing

Here we describe a new use case where the operator can optimize resource utilization by serving multiple bursty VMs with one NSM.

To make things concrete we draw upon a user traffic trace collected from a large cloud in September 2018. The trace contains statistics of tens of thousands of application gateways (AGs) that handle tenant (web) traffic in order to provide load balancing, proxy, and other services. The AGs are internally deployed as VMs by the operator. We find that the AG's average utilization is very low most of the time. Figure 7 shows normalized traffic processed by three most utilized AGs (in the same datacenter) in our trace with 1-minute intervals for a 1-hour period. We can clearly see the bursty nature of the traffic. Yet it is very difficult to consolidate their workloads in current cloud because they serve different customers using different configurations (proxy settings, LB strategies, etc.), and there is no way to separate the application logic with the underlying network stack. The operator has to deploy AGs as independent VMs, reserve resources for them, and charge customers accordingly.

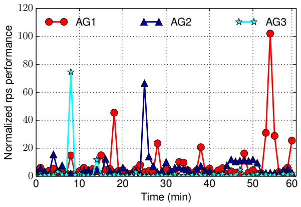


Figure 7: Traffic of three most utilized application gateways (AGs) in our trace. They are deployed as VMs.

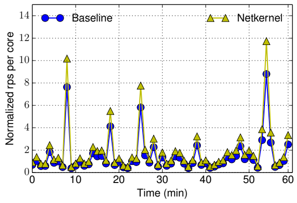


Figure 8: Per-core rps comparison. Baseline uses 12 cores for 3 AGs, while NetKernel with multiplexing only needs 9 cores.

NetKernel enables multiplexing across AGs running distinct services, since the common TCP stack processing is now separated into the NSM. Using the three most utilized AGs which have the least benefit from multiplexing as an example, without NetKernel each needs 4 cores in our testbed to handle their peak traffic, and the total per-core requests per second (rps) of the system is depicted in Figure 8 as Baseline. Then in NetKernel, we deploy 3 VMs each with 1 core to replay the trace as the AGs, and use a kernel stack NSM with 5 cores which is sufficient to handle the aggregate traffic. Totally 9 cores are used including CoreEngine, representing a saving of 3 cores in this case. The per core rps is thus improved by 33% as shown in Figure 8. Each AG has exactly the same rps performance without any packet loss.

In the general case multiplexing these AGs brings even more gains since their peak traffic is far from their capacity. For ease of exposition we assume the operator reserves 2 cores for each AG. A 32-core machine can host 16 AGs. If we use NetKernel with 1 core for CoreEngine and a 2-core NSM, we find that we can always pack 29 AGs each with 1 core for the application logic as depicted in Table 2, and the maximum utilization of the NSM would be well under 60% in the worst case for ∼97% of the AGs in the trace. Thus one machine can run 13 or 81.25% more AGs now, which means the operator can save over 40% cores for supporting this workload. This implies salient financial gains for the operator: according to [24] one physical core has a maximum potential revenue of $900/yr.

	Total Cores	NSM	CoreEngine	AGs
Baseline	32	0	0	**16**
NetKernel	32	2	1	**29**

Table 2: NetKernel multiplexes more AGs and saves over 40% cores.

6.2 Deploying mTCP without API Change

We now focus on use cases of deployment and performance benefits for users.

Most userspace stacks use their own APIs and require applications to be ported [4, 11, 34]. For example, in mTCP an application has to use `mtcp_epoll_wait()` to fetch events [34]. The semantics of these APIs are also different from socket APIs [34]. These factors lead to expensive code changes and make it difficult to use the stack in practice. The lack of modern APIs also makes it difficult to support complex web servers like nginx. mTCP also lacks some modern kernel TCP features such as advanced loss recovery, small queue, DSACK, *etc.*

With NetKernel, applications can directly take advantage of userspace stacks without any code change. To show this, we deploy unmodified nginx in the VM with the mTCP NSM we implement, and benchmark its performance using `ab`. Both VM and NSM use the same number of vCPUs. Table 3 depicts that mTCP provides 1.4x–1.9x improvements over the kernel stack NSM across various vCPU setting.

# vCPUs	1	2	4
Kernel stack NSM	71.9K	133.6K	200.1K
mTCP NSM	98.1K	183.6K	379.2K

Table 3: Performance of nginx using `ab` with 64B html files, a concurrency of 100, and 10M requests in total. The NSM and VM use the same number of vCPUs.

NetKernel also mitigates the maintenance efforts required from users. We provide another piece of evidence with mTCP here. When compiling DPDK required by mTCP on our testbed, we could not set the RSS (receive side scaling) key properly to the mlx5_core driver for our NIC and mTCP performance was very low. After discussing with mTCP developers, we were able to attribute this to the asymmetric RSS key used in the NIC, and fixed the problem by modifying the code in the DPDK mlx5 driver. We have submitted our fix to mTCP community. Without NetKernel users would have to deal with such technical complication by themselves. Now they are taken care of transparently, saving much time and effort for many users.

6.3 Shared Memory Networking

Inter-VM communication is well-known to suffer from high overheads [58]. A VM's traffic goes through its network stack, then the vNIC and the vSwitch, even when the other VM is on the same host. It is difficult for users and operator to optimize for this case, because a VM has no information about where the other endpoint is. The hypervisor cannot help either as the data has already been processed by the TCP/IP stack. With NetKernel the NSM is part of the infrastructure, the operator can easily detect the on-host traffic and use shared memory to copy data for the two VMs. We build a prototype NSM to demonstrate this idea: When a socket pair is detected as an internal socket pair by the GuestLib, and the two VMs belong to the same user, a shared memory NSM takes over their traffic. This NSM simply copies the message chunks between their hugepages and bypasses the TCP stack processing. As shown in Figure 9, with 7 cores in total, NetKernel with shared memory NSM can achieve 100Gbps, which is ~2x of Baseline using TCP Cubic and same number of cores.

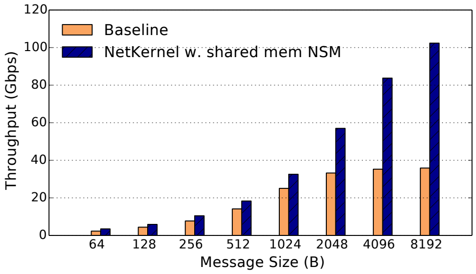


Figure 9: Using shared memory NSM for NetKernel for traffic between two colocated VMs of the same user. NetKernel uses 2 cores for each VM, 2 cores for the NSM, and 1 core for CoreEngine. Baseline uses 2 core for the sending VM, 5 cores for receiving VM, and runs TCP Cubic. Both schemes use 8 TCP connections.

7 Evaluation: Microbenchmarks

We now present microbenchmarks of crucial aspects of NetKernel: performance and multicore scalability in §7.2; isolation of multiple VMs in §7.3; and system overhead in §7.4.

7.1 Setup

Each of our testbed servers has two Xeon E5-2698 v3 16-core CPUs clocked at 2.3 GHz, 256 GB memory at 2133 MHz, and a Mellanox ConnectX-4 single port 100G NIC. Hyperthreading is disabled. We compare to the status quo where an application uses the kernel TCP stack in its VM, referred to as Baseline in the following. We designate NetKernel to refer to the common setting where we use the kernel stack NSM in our implementation. When mTCP NSM is used we explicitly mark the setting in the results. The same TCP parameter settings are used for both systems. The NSM uses the same number of vCPUs as Baseline since CPU is used almost entirely by the network stack in Baseline. NetKernel allocates 1 more vCPU for the VM to run the application and ServiceLib throughout the evaluation. Its CPU utilization is usually low: we report the actual CPU overheads of NetKernel in §7.4. The throughput results are measured by `iperf` and the rps results are measured by `ab`, unless stated otherwise. The throughput results are averaged over 5 runs each lasting 30 seconds.

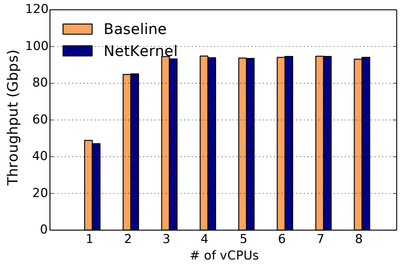


Figure 10: Send throughput of 8 TCP streams with varying numbers of vCPUs, 8KB messages.

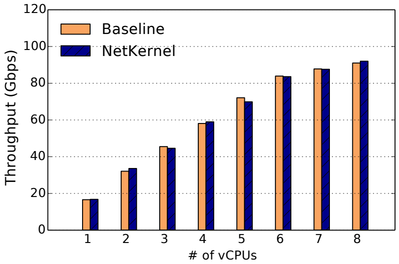


Figure 11: Recv throughput of 8 TCP streams with varying numbers of vCPUs, 8KB messages.

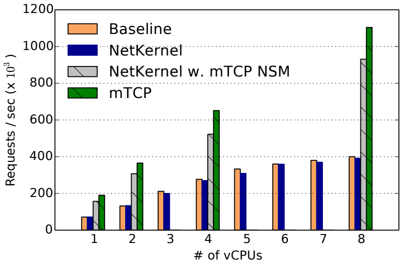


Figure 12: Performance of TCP short connections with multiple vCPUs. Message size 64B.

7.2 Performance and Scalability

We now look at NetKernel's basic performance.

NQE switching and memory copy. NQEs are transmitted by CoreEngine as a software switch. It is important that CoreEngine offers enough horsepower to ensure performance at 100G and higher. We measure CoreEngine throughput which is defined as the number of 32-byte NQEs copied from GuestLib's NK device queues to the ServiceLib's NK device queues. Table 4 shows the results with varying batch sizes. CoreEngine achieves ~8M NQEs/s without batching. With a small batch size of 4 or 8 throughput reaches 41.4M NQEs/s and 65.9M NQEs/s, respectively, which is sufficient for most applications.

We also measure the memory copy throughput between GuestLib and ServiceLib via hugepages. A memory copy in this experiment includes the following: (1) application in the VM issues a `send()` with data; (2) GuestLib gets a pointer from the hugepages; (3) copies the message to hugepages; (4) prepares a NQE with the data pointer; (5) CoreEngine copies the NQE to ServiceLib; and (6) ServiceLib obtains the data pointer and puts it back to the hugepages. Thus it measures the effective application-level throughput using NetKernel (including NQE transmission) without network stack processing.

We observe from Table 5 that NetKernel delivers over 100G throughput with messages larger than 4KB: with 8KB messages 144G is achievable. Thus NetKernel provides enough raw performance to the network stack and is not a bottleneck to the 100G deployment in production.

Batch Size (B)	1	2	4	8	16	32	64	128	256
NQEs per second ($\times 10^6$)	8.0	14.4	22.3	41.4	65.9	100.2	119.6	178.2	198.5

Table 4: CoreEngine switching throughput using a single core with different batch sizes.

Message Size (B)	64	128	256	512	1024	2048	4096	8192
Throughput (Gbps)	4.9	8.3	14.7	25.8	45.9	80.3	118.0	144.2

Table 5: Message copy throughput via hugepages with different message sizes.

Throughput. We examine throughput performance using the kernel stack NSM and 8 TCP streams with 8KB messages. Figures 10 and 11 show respectively the send and receive throughput with varying number of vCPUs. NetKernel achieves the same throughput performance and scalability with Baseline. The single-core send and receive throughput reaches 48Gbps and 17Gbps, respectively. Receive throughput is much lower because the kernel stack's RX processing is much more CPU-intensive with interrupts. Note that if the other cores of the NUMA node are not disabled, soft interrupts (softirq) may be sent to those cores instead of the one assigned to the NSM (or VM), thereby inflating the receive throughput. Both systems achieve the line rate of 100G using at least 3 vCPUs for send throughput as in Figure 10. For receive, both achieve 91Gbps using 8 vCPUs as in Figure 11.

Short TCP connections. We also benchmark NetKernel's performance in handling short TCP connections using a custom server sending a short message as a response. The server runs multiple worker threads that share the same listening port. Each thread runs an epoll event loop. Our workload generates 10 million requests in total with a concurrency of 1000. The connections are non-keepalive. The message size is 64B. Socket option `SO_REUSEPORT` is always used for the kernel stack. Figure 12 shows that NetKernel has the same multicore scalability as Baseline: performance increases from ~71Krps with 1 vCPU to ~400Krps with 8 vCPUs, i.e. 5.6x the single core performance. To demonstrate NetKernel's full capability, we also run the mTCP NSM with 1, 2, 4, and 8 vCPUs.[2] NetKernel with mTCP offers 167Krps, 313Krps, 562Krps, and 979Krps respectively, and shows better scalability than the kernel stack.

The results here show that NetKernel preserves the performance and scalability of network stacks, including high performance stacks like mTCP since our scalable queue design can ensure NetKernel is not the bottleneck and the contention is not severe in this situation.

7.3 Isolation

Isolation is important to ensure co-located users do not interfere with each other, especially in a public cloud. It is different from fair sharing: Isolation ensures a VM's performance guarantee is met despite network dynamics, while fairness ensures a VM obtains a fair share of the bottleneck capacity which varies dynamically. We conduct an experiment to verify NetKernel's isolation guarantees. As discussed in §4.4, CoreEngine

[2] Using other numbers of vCPUs for mTCP causes stability problems even without NetKernel.

uses round-robin to poll each VM's NK device for basic fairness. In addition, to achieve isolation we implement token buckets in CoreEngine to limit the bandwidth of each VM, taking into account varying message sizes. There are 3 VMs now: VM1 is rated limited at 1Gbps, VM2 at 500Mbps, and VM3 has unlimited bandwidth. They arrive and depart at different times. They are colocated on the same host running a kernel stack NSM using 1 vCPU. The NSM is given a 10G VF for simplicity of showing work conservation.

Figure 13 shows the time series of each VM's throughput, measured by our epoll server at 100ms intervals. VM1 joins the system at time 0 and leaves at 25s. VM2 comes later at 4.5s and leaves at 21s. VM3 joins last and stays until 30s. We can observe that NetKernel throttles VM1's and VM2's throughput at their respective limits correctly despite the dynamics. VM3 is also able to use all the remaining capacity of the 10G NSM: it obtains 9Gbps after VM2 leaves and 10Gbps after VM1 leaves at 25s. Therefore, NetKernel is able to achieve the same isolation in today's clouds with bandwidth caps.

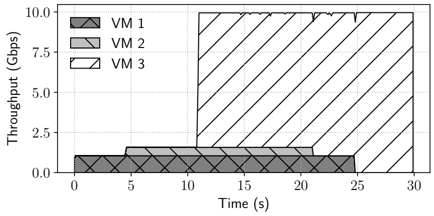


Figure 13: VM 1 is capped at 1Gbps, VM2 at 500Mbps, and VM3 uncapped. All VMs use the same kernel stack NSM. The NSM is assigned 10Gbps bandwidth. NetKernel isolates VM1 and VM2 successfully while allowing VM3 to obtain the remaining capacity.

7.4 Overhead

Latency. One may wonder if NetKernel with the NQE transmission would add delay to TCP processing, especially in handling short connections. Table 6 shows the latency statistics when we run `ab` to generate 1K concurrent connections to our epoll server for 64B messages. A total of 5 million requests are used. NetKernel achieves the same latency as Baseline. Even for the mTCP NSM, NetKernel preserves its low latency due to the much simpler TCP stack processing and various optimization [34]. The standard deviation of mTCP latency is much smaller, implying that NetKernel itself provides stable performance to the network stacks. We also investigate the latency without the effect of connection concurrency. To measure microsecond granularity latency, we use a custom HTTP client instead of `ab`, which reports application-level latency from the transmission of a request to the reception of the response. The experiments show the latency of Baseline and the NetKernel is 61.14μs and 89.53μs, respectively. The latency overhead is mostly introduced by CoreEngine in NetKernel.

CPU. Now to quantify NetKernel's CPU overhead, we use the epoll server at the VM side, and run clients from a different machine with fixed throughput or requests per second for both NetKernel and Baseline with kernel TCP stack. We disable all unnecessary background system services in both the VM and NSM, and ensure the CPU usage is almost zero without running epoll servers. During the experiments, we measure the total number of cycles spent by the VM in Baseline, and that spent by the VM and NSM together in NetKernel. We then report NetKernel's CPU usage normalized over Baseline's for the same performance level in Tables 7 and 8.

	Min	Mean	Stddev	Median	Max
Baseline	0	16	105.6	2	7019
NetKernel	0	16	105.9	2	7019
NetKernel, mTCP NSM	3	4	0.23	4	11

Table 6: Distribution of response times (ms) for 64B messages with 5 million requests and 1K concurrency.

Throughput	20Gbps	40Gbps	60Gbps	80Gbps	100Gbps
Normalized CPU usage	1.14	1.28	1.42	1.56	1.70

Table 7: Overhead for throughput. The NSM runs the Linux kernel TCP stack. We use 8 TCP streams with 8KB messages. NetKernel's CPU usage is normalized over that of Baseline.

Requests per second (rps)	100K	200K	300K	400K	500K
Normalized CPU usage	1.06	1.05	1.08	1.08	1.09

Table 8: Overhead for short TCP connections. The NSM runs the kernel TCP stack. We use 64B messages with a concurrency of 100.

We can see that to achieve the same throughput, NetKernel incurs relatively high overhead especially as throughput increases. To put things into perspective, we also measure CPU usage when the client runs in a docker container with the bridge networking mode. Docker incurs 13% CPU overhead compared to Baseline to achieve 40 Gbps throughout whereas NetKernel's is 28%. The overhead here is due to the extra memory copy from the hugepages to the NSM. It can be optimized away by implementing zerocopy between the hugepages and the NSM, which we are working on currently.

Table 8 shows NetKernel's overhead with short TCP connections. We can observe that the overhead ranges from 5% to 9% in all cases and is mild. As the message is only 64B here, the results verify that the NQE transmission overhead in NK devices is small.

8 Discussion

How can I do netfilter now? Due to the removal of vNIC and redirection from the VM's own TCP stack, some networking tools like netfilter are affected. Though our current design does not address them, they may be supported by adding additional callback functions to the network stack in the NSM. When the NSM serves multiple VMs, it then becomes challenging to apply netfilter just for packets of a specific VM. We argue that this is acceptable since most users wish to focus on their applications instead of tuning a network stack. NetKernel does not aim to completely replace the current architecture. Tenants may still use the VMs without NetKernel if they wish to gain maximum flexibility on the network stack implementation.

What about troubleshooting performance issues? In current virtualized environment, operators cannot easily determine whether a performance issue is caused by the guest network stack or the underlying infrastructure. With NetKernel operators gain much visibility of the guest network stack, which potentially facilitates debugging the performance issues. For example operators can closely monitor their NSMs to detect problems with the network stack; they can also deploy additional mechanisms in the NSMs to monitor their datacenter network [29, 49], all without disrupting users at all.

Does NetKernel increase the attack surface? It is well-known that shared memory design might suffer from side-channel attacks where malicious tenants could temper with other tenants' data on the hugepages. In this regard, NetKernel limits the visibility of NK devices into the hugepage for guest VMs: each device can only access its own address space. This is guaranteed by enforcing the address allocation and isolation control at CoreEngine.

How about supporting stacks with non-socket API? There are many fast network stacks with non-socket API such as PASTE [32], Seastar [11], and IX [18]. As NetKernel keeps the socket API, the central challenge to support these stacks (as NSMs) is how to resolve the semantic differences. While this requires case-by-case porting efforts, in general the ServiceLib should take care of the semantic transformation between the APIs.

Future directions. We outline a few future directions that require immediate attention with high potential: (1) Performance isolation. When multiple guest VMs share the same NSM, fine-grained performance isolation is imperative. In addition, it is necessary and interesting to design charging policies that promote fair use of the NSM and CoreEngine; (2) Resource efficiency. Various aspects of NetKernel's design can be optimized for efficiency and practicality. The CPU overhead of CoreEngine, mostly to poll the shared memory queues for NQE transmission, can be optimized by offloading to hardware like FPGA and SoC.

9 Related Work

We discuss related work besides those mentioned in §2.2.

There are many novel network stack designs to improve performance. The kernel stack continues to receive optimization in various aspects [42, 53, 64]. Userspace stacks based on fast packet I/O are also gaining momentum [7, 11, 34, 40, 45, 48, 55, 65]. Beyond transport layer, novel flow scheduling [16] and end-host based load balancing schemes [30, 37] are developed to reduce flow completion times. These proposals are targeting specific problems of the stack, and can be potentially deployed as NSMs in NetKernel. This paper takes on a broader and more fundamental issue: how can we properly re-factor the network stack, so that new designs can be easily deployed, and operating them in cloud can be more efficient?

Snap [47] is a microkernel networking framework that implements a range of network functions in userspace motivated by the need of rapid development and high performance packet processing in a private cloud. As NetKernel's design space and design choice are significantly different, it achieves many advantages that Snap does not target, such as multiplexing, porting a network stack across OSes or from kernel to user space, enforcing different network stack for different VMs, etc.

Lastly, our earlier position paper [51] presents the vision of network stack as a service. Here we provide the complete design, implementation, and evaluation of a working system in addition to several new use cases compared to [51].

10 Conclusion

We have presented NetKernel, a system that decouples the network stack from the guest, therefore making it part of the virtualized infrastructure in the cloud. NetKernel improves network management efficiency for operator, and provides deployment and performance gains for users. We experimentally demonstrated new use cases enabled by NetKernel that are otherwise difficult to realize in the current architecture. Through testbed evaluation with 100G NICs, we showed that NetKernel achieves the same performance and isolation as today's cloud.

We focused on efficiency benefits of NetKernel in this paper since they seem most immediate. The idea of separating network stack from the guest VM applies to public and private clouds as well, and brings additional benefits that are more far-reaching. For example, it facilitates innovation by allowing new protocols in different layers of the stack to be rapidly prototyped and experimented. It provides a direct path for enforcing centralized control, so network functions like failure detection [29] and monitoring [39, 49] can be integrated into the network stack implementation. It opens up new design space to more freely exploit end-point coordination [25, 54], software-hardware co-design, and programmable data planes [15, 43]. We encourage the community to fully explore these opportunities in the future.

Acknowledgment

We thank the anonymous ATC reviewers and our shepherd Michio Honda for their valuable comments. The project was supported in part by the Hong Kong RGC GRF (CityU Project #11210818). Dongsu was supported by MSRA Collaborative Research 2016 Grant Award. Keith was supported by a Sloan Research Fellowship and by Google, Huawei, VMware, Dropbox, Amazon, and Facebook.

References

[1] Amazon EC2 Container Service. https://aws.amazon.com/ecs/details/.

[2] Azure Container Service. https://azure.microsoft.com/en-us/pricing/details/container-service/.

[3] Busy Polling: Past, Present, Future. https://netdevconf.info/2.1/papers/BusyPollingNextGen.pdf.

[4] F-Stack: A high performance userspace stack based on FreeBSD 11.0 stable. http://www.f-stack.org/.

[5] Google container engine. https://cloud.google.com/container-engine/pricing.

[6] Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx.html.

[7] Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. http://www.moderntech.com.hk/sites/default/files/whitepaper/V10_Solarflare_OpenOnload_IntroPaper.pdf.

[8] Mellanox Smart Network Adaptors. http://www.mellanox.com/page/programmable_network_adapters?mtag=programmable_adapter_cards.

[9] Netronome. https://www.netronome.com/.

[10] Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. http://caxapa.ru/thumbs/793343/Open_Source_Kernel_Enhancements_for_Low-.pdf.

[11] Seastar. http://www.seastar-project.org/.

[12] mTCP. https://github.com/eunyoung14/mtcp/tree/2385bf3a0e47428fa21e87e341480b6f232985bd, March 2018.

[13] The Open Group Base Specifications Issue 7, 2018 edition. IEEE Std 1003.1-2017. http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html, 2018.

[14] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[15] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. HotCocoa: Hardware Congestion Control Abstractions. In *Proc. ACM HotNets*, 2017.

[16] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proc. USENIX NSDI*, 2015.

[17] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[18] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. USENIX OSDI*, 2014.

[19] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *Commun. ACM*, 60(2):58–66, February 2017.

[20] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized Congestion Control. In *Proc. ACM SIGCOMM*, 2016.

[21] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proc. USENIX NSDI*, 2018.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. ACM SOSP*, 1995.

[23] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proc. NSDI*, 2017.

[24] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. USENIX NSDI*, 2018.

[25] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. ACM CoNEXT*, 2015.

[26] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Proc. the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[27] A. Greenberg. SDN in the Cloud. Keynote, ACM SIGCOMM 2015.

[28] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proc. ACM CoNEXT*, 2010.

[29] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proc. ACM SIGCOMM*, 2015.

[30] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*, 2015.

[31] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proc. ACM SIGCOMM*, 2016.

[32] M. Honda, G. Lettieri, L. Eggert, and D. Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *Proc. USENIX NSDI*, 2018.

[33] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX NSDI*, 2014.

[34] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. USENIX NSDI*, 2014.

[35] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proc. USENIX NSDI*, 2013.

[36] G. Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *Proc. USENIX NSDI*, 2015.

[37] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford. CLOVE: How I Learned to Stop Worrying About the Core and Love the Edge. In *Proc. ACM HotNets*, 2016.

[38] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella. Iron: Isolating Network-based CPU in Container Environments. In *Proc. USENIX NSDI*, 2018.

[39] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *Proc. USENIX NSDI*, 2019.

[40] D. Kim, T. Yu, H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *Proc. USENIX NSDI*, 2019.

[41] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *Proc. USENIX HotCloud*, 2014.

[42] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proc. ASPLOS*, 2016.

[43] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multi-Path Transport for RDMA in Datacenters . In *Proc. USENIX NSDI*, 2018.

[44] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ASPLOS*, 2013.

[45] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proc. ACM SIGCOMM*, 2014.

[46] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI*, 2014.

[47] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A microkernel approach to host networking. In *Proc. ACM SOSP*, 2019.

[48] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM*, 2015.

[49] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proc. SIGCOMM*, 2016.

[50] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring Endpoint Congestion Control. In *Proc. ACM SIGCOMM*, 2018.

[51] Z. Niu, H. Xu, D. Han, P. Wang, and L. Liu. Netkernel: Network stack as a service in the cloud. In *Proc. ACM HotNets*, 2017.

[52] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*, 2016.

[53] S. Pathak and V. S. Pai. ModNet: A Modular Approach to Network Stack Extension. In *Proc. USENIX NSDI*, 2015.

[54] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proc. ACM SIGCOMM*, 2014.

[55] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. USENIX OSDI*, 2014.

[56] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Proc. ACM SIGCOMM*, 2012.

[57] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. In *Proc. ACM SIGCOMM*, 2013.

[58] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *Architectures for Networking and Communications Systems*, 2013.

[59] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *Proc. ACM SIGCOMM*, 2017.

[60] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. USENIX NSDI*, 2011.

[61] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. USENIX OSDI*, 2010.

[62] J. Son, Y. Xiong, K. Tan, P. Wang, Z. Gan, and S. Moon. Protego: Cloud-Scale Multitenant IPsec Gateway. In *Proc. USENIX ATC*, 2017.

[63] B. Stephens, A. Singhvi, A. Akella, and M. Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proc. USENIX ATC*, 2017.

[64] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proc. USENIX ATC*, 2016.

[65] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proc. ACM SIGCOMM*, 2015.

[66] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proc. USENIX NSDI*, 2019.

Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services

Mingyu Wu[†‡], Ziming Zhao[†‡], Yanfei Yang[†‡], Haoyu Li[†‡], Haibo Chen[†‡], Binyu Zang[†‡], Haibing Guan[†‡], Sanhong Li[◇], Chuansheng Lu[◇], Tongbao Zhang[◇]

†*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
‡*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
◇*Alibaba Group*

Abstract

The service-oriented architecture decomposes a monolithic service into single-purpose services for better modularity and reliability. The interactive nature, plus the fact of running inside a managed runtime, makes garbage collection a key to the reduction of tail latency of such services. However, prior concurrent garbage collectors reduce stop-the-world (STW) pauses by consuming more CPU resources, which can affect the application performance, especially under heavy workload.

Based on an in-depth analysis of representative latency-sensitive workloads, this paper proposes *Platinum*, a new concurrent garbage collector to reduce the tail latency with moderate CPU consumption. The key idea is to construct an isolated execution environment for concurrent mutators to improve application latency without interfering with the execution of GC threads. *Platinum* further leverages a new hardware feature (i.e., memory protection keys) to eliminate software overhead in previous concurrent collectors. An evaluation against state-of-the-art concurrent garbage collectors shows that *Platinum* can significantly reduce the tail latency of real-world interactive services (by as much as 79.3%) while inducing moderate CPU consumption.

1 Introduction

Today's cloud environment is an enormous beast with quantities of interconnected machines. To tame the beast, developers (1) break the traditional monolithic applications into small and interactive *services* and (2) implement services atop managed languages (such as Java, C#, and Go) to build reliable, elastic and efficient systems. Unfortunately, a tension exists between those services and managed languages. Services are designed as single-purpose and interactive applications to achieve low latency. It typically takes several milliseconds and even sub-millisecond to complete a request in interactive services. Meanwhile, managed languages like Java introduce garbage collections (GC) to manage memory resources automatically. However, mainstream garbage collectors used in interactive services will introduce stop-the-world (STW) events, where all application threads (known as *mutators*) are forced to pause so that GC threads can scan the heap for memory reclamation. The pause time is tens to hundreds of milliseconds, which are usually one to two magnitudes of the execution time for a single request in interactive services. Therefore, STW pauses will affect the latency of services and cause the long tail problem. Prior work has observed that STW pauses have significant effects on tail latency in latency-sensitive scenarios [19, 40].

There are mainly two ways to reduce the STW pause time. *Partially-concurrent collectors*, such as G1 [11] and CMS, suggests reducing STW pauses by restricting the size of *collection set* in which objects need to be collected. However, this solution will introduce more frequent collections and larger accumulated STW pause time. *Mostly-concurrent collectors*, such as Shenandoah [13] and ZGC [33], allow mutators to run in nearly all GC phases. Those collectors are quite effective in reducing the duration of STW pauses, but it requires GC threads to run constantly and spend more computing resources coordinating with mutators. In summary, both kinds of collectors achieve shorter STW pauses by occupying more CPU slices and thereby put more pressure on mutators. When the workload becomes stressful, spending more computing resources in GC may end up with even worse application latency.

In this paper, we present a new garbage collector which (1) reduces the tail latency of interactive services and (2) induces moderate CPU consumption. To achieve this, we first study the effect of GC on various latency-sensitive interactive service scenarios, including production traces in Alibaba, whose business applications rely heavily on JVM. We further analyze state-of-the-art collectors and uncover their problems respectively: *idle computing resources* in stop-the-world pauses of partially-concurrent collectors and *considerable runtime overhead* in mostly-concurrent collectors. We then describe the *skewed memory write* behaviors of interactive service applications. We also discuss the development of hardware to show opportunities for a brand-new design.

According to the analysis, this paper proposes *Platinum*, which finds a sweet-spot in the design of concurrent collectors. It leverages idle cores in STW pauses and grants them to mutators to solve the idle computing resources problem. It then provides an isolated execution model to minimize the synchronizations between GC threads and mutators to reduce the runtime overhead in prior mostly-concurrent collectors. It further exploits recently-released hardware features (MPK) to eliminate *barriers*, the primary source of software overhead in traditional mostly-concurrent collectors. With *Platinum*, GC threads and mutators can run together with little interference with each other, so the latency and CPU utilization are both satisfying.

Platinum is implemented atop the Parallel Scavenge Garbage Collector (PSGC), a STW and throughput-oriented collector used by default in the HotSpot JVM of OpenJDK 8. Evaluation of various interactive service benchmarks show that *Platinum* significantly improves the tail latency of applications (by up to 79.3% for 99th percentile latency) compared with other concurrent collectors while preserving moderate CPU utilization.

The contributions of this paper include:

- A comprehensive analysis of latency-sensitive interactive services, including simulated industrial workload with production traces, to understand the effect of GC (Section 2).
- *Platinum*, a concurrent garbage collector that can reduce the tail latency for interactive services while preserving moderate CPU consumption (Section 4).
- Experiments on different latency-sensitive scenarios to confirm that *Platinum* can outperform other garbage collectors on tail latency and CPU utilization for interactive services (Section 5).

2 Analysis: when interactive services meet GC

In this section, we will analyze the effect of GC in interactive services with production traces in Alibaba.

2.1 A page is multiple services

Alibaba has one of the world's largest e-commerce platforms. To serve an ocean of concurrent requests from a vast number of clients at any time, Alibaba has deployed its platform atop a large scale of machines. Traditional monolithic applications are too cumbersome and thus not suitable to be distributed to many machines due to the prohibitive cost of development and maintenance, so the developers from Alibaba have split them into smaller, simple-purpose, and interactive units, namely *services*. A service is much smaller to simplify the deployment process, and it can also be replicated to enhance the availability of the overall platform.

Due to the complex business logic in Alibaba, every operation from users require the collaboration of various services. For example (shown in Figure 1), when a user wants to check out, she will request for the check-out page, where all items in the shopping cart (cart service) will be combined together so that the most cost-effective way to purchase them with available coupons will be automatically computed (coupon service). In addition, the check-out page also recommends other items according to those in the cart and the user's prior purchase behaviors (recommendation service). Those services also interact with each other, and they may communicate with the cache service for high-speed data fetching.

Since those services are mostly written in Java for reliability, productivity, and compatibility, all of them will be affected by GC. Alibaba has provided some workarounds to mitigate the effect of GC. For example, when the recommendation service is not responsive, the page render is capable of generating a simplified web page without any recommendation information for users. Unfortunately, not all services can benefit from those optimizations. For example, as for the coupon service, the latest-available coupons must be included for computation. Otherwise, the users will fail to purchase in the most cost-effective way. We have conducted a series of tests to understand the role GC plays in those latency-critical services. Note that a garbage collector usually includes *minor GC* and *major GC*. The minor GC collects a part of the heap while the major GC collects the whole heap. Since major GC rarely happens in the scenario of interactive services, this work will focus on the effect of minor GC.

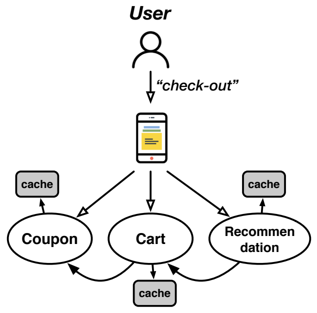


Figure 1: A simplified model to shape the multi-services scenario in Alibaba

2.2 STW pauses: the culprit for tail latency

Since garbage collectors will pause application threads for memory reclamation, it will significantly affect the response time of requests in interactive services. To better understand the effect of GC, we leverage a simulated online environment for analysis. The simulated environment is built with 120 service instances, each of which is deployed in a container. All services are unmodified applications extracted from the real industrial workload in Alibaba. During testing, those services will be fed with requests at a given throughput. In our setting, the cluster will handle 200 user requests per second, and this value is chosen to stress the coupon services. All requests are traces collected from the production environment. The default garbage collector for services is CMS (Concurrent Mark Sweep [28]), a classic concurrent collector introducing relatively long STW pauses.

To understand the relationship between application behavior and GC pause time, we add a new Java option -*XX:InstrumentedPauseTime* to the vanilla OpenJDK 8. When the value is not zero, JVM will add a *sleep* call at the end of GC to extend the GC pause time. The duration of sleeping time can be adjusted by modifying the value of -*XX:InstrumentedPauseTime*. We are mainly interested in the coupon service as it is latency-sensitive. Our findings are listed below.

Stop-The-World (STW) pauses is a killer factor for the tail latency. We exploit a scatter plot in Figure 2 to illustrate the relationship between GC and request latency. Points in the scatter plot stand for the completion time of a request (measured in the server-side), while red vertical lines stand for the start time of GC. As Figure 2 suggests, each GC cycle will follow some stragglers, which significantly affect the tail latency. Although the request latency could be influenced by many factors such as network, disk I/O, and other collaborative services, GC is the one to dominate the tail latency. Note that this experiment actually underestimates the effect of GC on the request latency, as the statistics are collected from servers, which overlook the queuing time in the client-side. One can expect more requests are affected by GC if each request can be tracked in the upstream services, which is unfortunately not supported in the cluster environment.

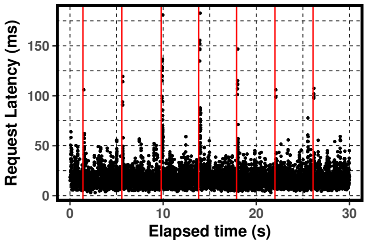


Figure 2: The relationship between STW pauses and request latency in the coupon service. The collector is CMS

GC pause time shows a super-linear relationship with tail latency. During the evaluation, we change the value of -*XX:InstrumentedPauseTime* for all five coupon service instances from 0 to 10ms and collect the log from them. Figure 3 shows the change in the request latency for a coupon service instance in a cumulative distribution function (CDF) form, compared with that in the vanilla setting. The result suggests that the increased pause time has a magnified impact on the tail latency: When the GC pause is added by 10 ms, the 99th percentile latency is increased by 11.435 ms, and the 99.9th percentile latency is enlarged by 32.950 ms. It is because requests in clients are generated at any time, regardless of the running state of services. When the coupon service instance is undergoing a garbage collection, the pending requests will gradually increase and queue up. Once GC ends, a pending request cannot be processed until the preceding ones are finished. If GC pauses are extended, those newcomers must wait for not only a longer pause but also more pending requests, so the tail latency will increase faster than GC pauses.

In the real-world cooperative multi-services scenario (such as the check-out case in Alibaba), the problem can be deteriorated because all participating services may be affected by GC. For example, Mass et al. [25] have observed that Cassandra replicas on GC will hinder the whole storage system from constructing a quorum, which leads to prohibitive user-experienced latency. Therefore, GC pauses is a key factor for tail latency reduction in interactive services.

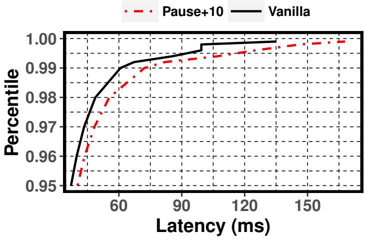


Figure 3: The CDF of request latency for the coupon service atop CMS

2.3 Is concurrent GC helpful?

Due to the detriment of STW pauses, interactive services usually adopt *concurrent GC* for better application latency. Concurrent GC can be roughly divided into two categories: *partially-concurrent collectors* and *mostly-concurrent collectors*. Partially-concurrent collectors, such as CMS (mentioned above) and G1 [11], allow the co-execution for mutators and GC threads in only some phases of GC. Therefore, they still introduce STW pauses, and they provide solutions to further reduce them. *Mostly-concurrent collectors*, such as ZGC [33] and Shenandoah [13], nearly eliminate STW pauses so that mutators can run constantly. We study those two kinds of collectors on interactive services respectively.

Partially-concurrent GC. As shown in Figure 2, the tail latency problem exists in partially concurrent collectors like CMS since they still pause mutators for collection. Fortunately, partially-concurrent collectors usually provide options to adjust the duration of pauses. For example, CMS allows users to adjust the size of the heap area required to be collected, while G1 can be restricted with a pre-assigned maximum pause time. After setting those options, partially-concurrent collectors will adjust the heap layout to meet the requirement.

We exploit G1 to study the effect of those adjustable options. G1 is a highly-tunable partially-concurrent collector which becomes the default one since OpenJDK 9. It is a generational collector whose heap space consists of *young space* and *old space*. Its GC algorithm contains three parts: *minor GC* on the young space, *mixed GC* on the young space and a part of the old space, and *major GC* on the whole heap. The minor GC, which is stop-the-world, happens the most frequently. To reduce the duration of pauses, G1 provides an option *-XX:MaxGCPauseMillis* for users to control them. Therefore, we launch the coupon service on G1 by setting the option to various values (30ms, 40ms, 60ms) and evaluate the performance respectively. This evaluation is single-point, where only one coupon service is launched to process requests. With the single-point evaluation, the latency can be accurately collected from the client-side, including the aforementioned queuing delay. The requests are sent in a fixed throughput to simulate a stressful scenario (4000 requests per second in our setting), and they are still real-world traces extracted from the online environment in Alibaba. The duration for data collection is a minute.

Table 1 shows the statistics on GC and application for different settings. With smaller *MaxGCPauseMillis*, both the minimum and the average GC pause time are reduced. However, young GC is triggered much more frequently. It is because G1 controls the GC time by tuning the size of the young space, which serves for memory allocation requests from mutators. Since young GC is triggered when the memory resource in the young space is exhausted, its frequency will increase when the young space shrinks. As a result, although the per-GC pause time is cut down by lowering *MaxGCPauseMillis*, the overall time consumed by GC grows larger. With larger overall GC pause time, more application requests are affected, and less computing resource is available for mutators. Therefore, the tail latency problem is not resolved but becoming much more severe: the p99 latency with the 30ms setting is 13X of that with 60ms. Besides, the average CPU utilization in the 30ms setting is also increased by 15.3% compared with the 60ms setting. This experiment suggests that partially-concurrent GC achieves shorter pauses by consuming more CPU resources, which may end up with worse tail latency.

Metrics	30ms	40ms	60ms
Minimum GC pause (ms)	21.815	21.459	39.856
Average GC pause (ms)	34.441	40.724	48.491
The number of GC	550	392	111
p99 latency (ms)	1942.09	1389.99	148.85
Average CPU utilization	51.45%	50.81%	36.17%

Table 1: The statistics on GC and the coupon application with different settings of G1GC

Mostly-concurrent GC. Compared with partially-concurrent GC, mostly-concurrent GC allows mutators to execute nearly all the time. Recently released mostly-concurrent collectors, like ZGC and Shenandoah, claim to have reduced GC pauses to several milliseconds regardless of the heap size. In this work, we mainly study Shenandoah[1], a mostly-concurrent garbage collector released in OpenJDK 12.

We launch the coupon service atop Shenandoah and evaluate it with the same setting as G1. After collecting the GC log, we conclude that Shenandoah is very effective in reducing the pauses, and the average pause time is only 18.764 ms. However, the latency of requests is prohibitive: the p99 latency is over 3 seconds, which is 1.86X even compared with the worst case in G1 (30 ms). We observed that the time when GC threads are active is 53.05 seconds, which means that GC threads are active nearly all the time. Meanwhile, the average CPU utilization reaches 83.05% (the peak utilization reaches 96.74%), which suggests that GC threads consume much more CPU resources than other collectors, and mutators do not have enough CPU slices to sustain such a high throughput. As a result, the p99 latency with Shenandoah is not decreased but increased when encountering stressful workload.

To conclude, both partially-concurrent GC and mostly-concurrent GC are making a tradeoff between the duration of GC pauses and the CPU efficiency. Shorter GC pauses mean that GC threads will consume more computing resources and thus affect the performance of mutators. We instead want to build a garbage collector with both short pauses and moderate CPU efficiency to support the interactive services.

3 Implications for a new GC design

Before proposing our design, we take a closer look at the designs of prior concurrent garbage collectors to uncover the opportunities to build a new garbage collector for our goals: short GC pauses and moderate CPU utilization. We then reveal the *skewed memory write* behavior in interactive service applications, which is crucial to our design. We also introduce *memory protection keys* (MPK), a recently released hardware feature, and suggest how collectors can benefit from it.

3.1 Problems in concurrent garbage collectors

According to the behavior of mutators during GC, concurrent garbage collectors can be roughly divided into two categories. However, both of them have problems hindering them from achieving both satisfying GC pause time and CPU efficiency.

Idle computing resources for partially-concurrent collectors. STW pauses in partially-concurrent collectors require all mutators to suspend and leverage all computing resources to collect objects. This design avoids interferences between mutators and GC threads, but it also results in idle cores during GC due to its scalability issues.

There are two reasons why garbage collectors cannot scale well. First, the collection algorithm is somewhat similar to

[1] We exploit Shenandoah as a baseline in this paper as it provides backward support to OpenJDK 8, a long-time-support version which is widely leveraged in both industries (like Alibaba) and open-sourced projects.

graph traversal: GC threads will start from some *root objects* and mark all reachable ones through references among objects. The traversal is highly unpredictable as we do not know how many references a thread will process in advance. Therefore, collectors are prone to load-imbalance and often turns to work-stealing to achieve dynamic balancing. However, work-stealing is also ineffective in that it has to search for tasks from all other threads and contend with others during task fetching. Prior work [37] shows that work-stealing even causes performance slowdown in extreme cases. Second, the scalability of collectors is affected by many factors. Recent studies have shown that the NUMA architecture [17], synchronization protocols [16], and even the Linux scheduling mechanism [38, 40] could have a significant influence on the GC scalability. Therefore, it is difficult to come up with a general algorithm which is scalable for various scenarios.

Given those reasons, partially-concurrent collectors exploit a conservative mechanism where the number of GC threads is smaller than that of cores. In OpenJDK, the number of GC threads by default is about five-eighths of the total core count. This policy also works for STW collectors like PSGC. The developers of OpenJDK explain their choice in the comment of the source code: *For very large machines, there are diminishing returns for large numbers of worker threads. Instead of hogging the whole system, use a fraction of the workers for every processor after the first 8.* This default setting mitigates performance degradation as the number of cores increases and has been used in prior GC studies [40]. However, it also results in idle cores during collection. Instead of searching for a perfectly scalable collection algorithm, we want to introduce some concurrent mutators to leverage those idle cores while still preserving the performance of collectors. Note that introducing concurrent mutators during GC will not hurt the overall CPU efficiency much, as the duration of STW pauses only occupies a very small portion of the whole application's execution time in interactive services.

Considerable runtime overhead for mostly-concurrent collectors. Unlike partially-concurrent collectors, mostly-concurrent ones allow mutators to run simultaneously with GC threads nearly all the time to reduce the application latency. However, GC threads and mutators must synchronize with each other as they may modify the same objects simultaneously, which introduces overhead for both GC threads and mutators. Besides, mostly concurrent collectors further introduce *barriers* in mutators for GC invariant checking. A barrier is a piece of code instrumented before specific instructions. For example, Shenandoah exploits *read barriers* for mutators, meaning that mutators need to check the invariants for every single read operation on references, no matter if GC is active. Those two factors introduce significant runtime overhead to the runtime. Therefore, although Shenandoah has been optimized through aggressive Just-In-Time (JIT) compilation, it still causes a 24% slowdown for real-world workload compared with G1 [13].

3.2 The skewed memory write behavior

Memory behavior of applications is quite crucial to GC performance and thereby affect the design of collectors. For example, the memory behavior of big-data processing workload is at odds with assumptions in traditional GC algorithms and stimulates a series of big-data-friendly garbage collectors [18, 31]. To this end, we study the memory behavior of latency-sensitive services to explore new space for garbage collector design.

Session-based execution model. Since services are interactive, their execution can be divided into many *sessions*. The service will process a request when a session starts and generate a response before the session ends. Sessions are mostly isolated from each other: a session will not try to access the objects created by other sessions unless those objects are globally visible through shared data structures. Therefore, we presume that the memory behavior of those session-based applications will be skewed, i.e., *the memory accesses within a session will fall into a very small range where the session allocates memory.* The memory range is referred to as *working set* in this paper. We have conducted an in-depth analysis to validate this hypothesis.

Memory behavior analysis. To analyze the memory behavior of those session-based applications, we first need to demarcate all sessions. We add two new JVM calls, *Sessionbegin* and *Sessionend*, to achieve this goal. After *Sessionbegin* is invoked, the JVM will track all memory the session allocates. When *Sessionend* is called, the JVM will print out the size of the overall allocated memory, which is the working set size for the current session.

Our hypothesis is that session-based interactive services share similar memory behavior. To confirm this, we have also studied three other applications: *SpecJBB2015*, a simulated online supermarket, *Cassandra* [9], a key-value store, and *ShopCenter*, another interactive service used in Alibaba. All of them will process requests from clients in sessions and send responses back. Note that both ShopCenter and Coupon leverage production traces for analysis. To profile their memory behavior, we instrument *Sessionbegin* right before the request is processed and *Sessionend* when the processing finishes.

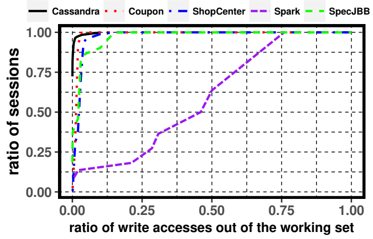


Figure 4: The CDF for the memory range of write accesses in various applications

Figure 4 shows the CDF curve of memory writes for three different interactive applications. As for the coupon service, over 99.5% sessions have less than 3% writes out of their working sets. The working set of a session usually spans several megabytes, which is quite small compared to the Java heap (typically tens to hundreds of gigabytes). The other applications share similar memory behaviors. For Cassandra, even the worst case has 72.4% writes inside its own working set. Other kinds of applications, however, do not follow this behavior. As for Spark, a big-data processing framework, 50% sessions (a session in Spark is defined as the process the worker handles a task assigned by the master) have more than 46% writes out of their working sets.

This experiment confirms our hypothesis that memory writes in sessions of interactive services are skewed. The skewed memory write behavior opens up an opportunity for optimization: since mutators mostly update objects inside their working sets, they can run simultaneously aside GC threads with rare interference if GC threads avoid reclaiming their working sets. Since previous garbage collectors do not put the skewed memory write behavior into consideration, this finding motivates us to design a new garbage collector to guarantee both satisfying latency and CPU utilization.

3.3 MPK and Garbage Collectors

Since barriers in concurrent GC are costly due to its high execution frequency, some collectors [2, 4, 10] have turned to a virtual-memory-based mechanism for a transparent and efficient solution. The virtual-memory-based mechanism leverages the access permissions on the page table entries and relies on hardware to check the invariants. For example, a GC thread can mark a virtual page as *being-collected* by setting its permission as read-only. Mutators writing to this page will trigger a page fault and execute synchronization-related operations in a pre-registered handler. GC threads working on other pages will have no overhead as no page fault is triggered. This method can eliminate the need for the barrier code, and it can also be used in other areas, such as software transactional memory [1].

Unfortunately, threads in a process will share the same permissions on all page table entries. In the previous example, if a GC thread also wants to modify the being-collected page, which should be legal, it still suffers from a page fault. Those *false page faults* impede collectors to implement an efficient protection mechanism.

Intel MPK (Memory Protection Keys) is a hardware feature available recently in the SkyLake server CPUs. With MPK, users can categorize virtual pages into different *domains*, which will be denoted with special bits in the corresponding page table entries. Furthermore, they can manually grant different permissions for each domain to different threads through a special register. This hardware feature makes it possible to support finer-grained *thread-level isolation* by adjusting the permissions over domains for each thread. Previous work has studied the usage of MPK in the security area [14, 20, 35, 43], but it could also be leveraged for performance consideration.

4 Design

According to our analysis, we build *Platinum* for both satisfying latency and CPU efficiency.

4.1 Overview

Platinum is built atop PSGC, a STW garbage collector in OpenJDK. Compared with concurrent collectors, PSGC is more CPU-efficient as it always pauses mutators during GC to achieve maximum collection throughput. Nevertheless, it still results in idle cores due to scalability issues. *Platinum* inherits the generational design from PSGC to divide the heap into *young space* and *old space*. The young space further consists of three sub-spaces. The *eden-space* (usually the largest sub-space) serves for allocation, while the other twos, *from-space* and *to-space*, are used to store objects surviving at least one GC cycle. Only objects having lived for long will be copied into *old space*. It contains two GC algorithms: minor GC for the young space and major GC for the whole heap. *Platinum* mainly considers the minor GC while leaving the major GC as future work.

Figure 5 illustrates the infrastructure of *Platinum*. The design highlights of *Platinum* include:

Sufficiently leveraging computing resources. *Platinum* collects the cores not used by GC threads and grants them to mutators for better application latency, hence resolving the idle computing resources problem in partially-concurrent GC.

Isolated execution between GC threads and mutators through heap partition. According to the skewed memory behavior in interactive services, *Platinum* partitions the heap space so that GC threads and mutators can focus on processing objects in different parts of the heap. This design minimizes the synchronizations between GC threads and mutators, so the coordination overhead in mostly-concurrent collectors is greatly reduced in *Platinum*.

Hardware-assisted barrier elimination. *Platinum* leverages the MPK hardware feature to remove the need for software barriers adopted in prior mostly-concurrent collectors. This design further reduces the runtime overhead and improves CPU efficiency. *Platinum* also exploits the RTM feature to ensure the atomicity of write operations in mutators.

4.2 Platinum in steps

Similar to the minor GC algorithm in PSGC, *Platinum* is copy-based: GC threads will simultaneously scan the young space for live objects and copy them to their new address. The process of *Platinum* GC is mainly in three steps:

1. Initial marking. This step is somewhat similar to the initial marking pause in G1 [11]. In the initial marking phase of *Platinum*, GC threads will scan the runtime stack of muta-

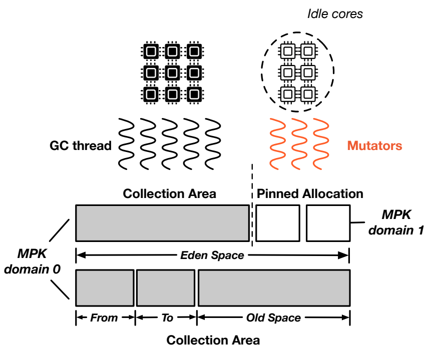


Figure 5: Overview of *Platinum*

tors to mark live objects. Those live objects on the stack will be treated as *root objects* for further object copying. This step is stop-the-world because the values on the stack are changed quite frequently, and we want a stable state at the beginning of a collection. It usually takes only a few milliseconds to finish initial marking.

2. Concurrent scavenge. After initial marking, *Platinum* will invoke mutators to resume their execution while GC threads will concurrently scan the heap to identify and copy live objects. Thanks to the isolated execution mechanism (detailed in Section 4.4), GC threads and mutators will focus on processing objects in different areas of the heap and hardly interfere with each other. Therefore, GC threads can directly copy live objects without considering the behavior of mutators. This step occupies the most time of the whole GC process in *Platinum*.

3. Stop-the-world scavenge. When GC threads have finished their work, *Platinum* will pause running mutators again. Since some objects in mutators' working sets are not processed by GC threads in the concurrent scavenge step, they may contain "stale" references to objects which have been evacuated. Therefore, *Platinum* should scan those objects for correctness guarantee. Since the number of objects is quite small, this step will not take long.

With the above three steps, *Platinum* can: (1). improve the application latency because mutators are allowed to run in most time of GC; (2). avoid consuming too much CPU resources because GC threads are isolated from mutators to retain satisfying collection throughput. Therefore, *Platinum* can take into account both latency and CPU efficiency at the same time. We will introduce the core designs of *Platinum* in the rest of this section.

4.3 Idle core collection

To make *Platinum* effective, the application should configure the number of GC threads to be smaller than that of cores (or directly adopt the default setting in OpenJDK). When *Platinum* is initialized, it will automatically bind the GC thread into different CPU cores. Other cores with no GC threads running will be remembered by *Platinum* as *idle cores*.

When *Platinum* is not active, mutators are free to run on any cores. When GC is triggered, *Platinum* will constrain mutators to only choose idle cores for execution to avoid contending computing resources with GC threads. This is achieved by setting the affinity values of mutators with the *sched_setaffinity* interface in Linux. Those affinity values will be reset at the end of GC. This design ensures not only idle cores are sufficiently used by mutators but also each GC thread monopolizes its assigned core.

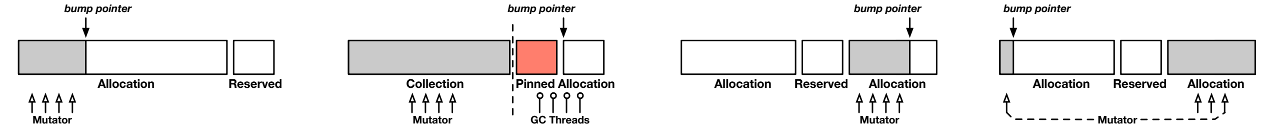
4.4 Isolated execution with heap partition

To achieve isolated execution between GC threads and mutators, *Platinum* partitions the heap into three *areas* during GC (shown in Figure 5). The first area, namely *collection area*, will be collected by GC threads. The collection area covers the most part of Java heap, including from-space, to-space, old-space, and the largest part of eden-space.

The other two areas, in contrast, are used by mutators and thus not collected in this GC cycle. Since write operations of mutators fall into a very small range (see Section 3.2), we use the *pinned area* to include objects which are highly possible to be modified by mutators in the near future. The last part is the *allocation area*, which is used by mutators to allocate new objects during garbage collection. Those newly allocated objects should be considered alive and only be scanned in the next collection cycle.

Figure 6 illustrates how the three areas work in *Platinum*. During normal execution, *Platinum* will partition the eden-space into two areas (Figure 6a). The larger one will serve memory allocation requests from mutators while the smaller one will be reserved. *Platinum* also adopts a *bump pointer* to denote how much memory has been used.

When the bump pointer reaches the end of the allocation area, GC will be triggered, and GC threads will become active. In the initial marking phase, *Platinum* will partition the eden-space into the three areas mentioned before (Figure 6b). The reserved area in the normal execution will become the *allocation area* where concurrent mutators create new objects during GC. The larger one will be further split into *collection area* and *pinned area*. GC threads will only collect the collection area while mutators mainly modify the pinned area and the allocation area. Compared with the collection area, the pinned area resides close to the bump pointer. This design choice is based on the memory allocation mechanism in JVM: each mutators will first allocate a *segment* from the global heap, and then allocate memory from the segment until it is filled up. Therefore, *Platinum* locates the pinned area adjacent to the allocation area to contain the latest segments allocated by different threads. The size of the pinned area is preset to a fixed proportion of the whole eden-space. The default value is 1/128, which can include segments from tens of mutators while inducing moderate pauses during the stop-the-world scavenge step. *Platinum* also allows users to tune this value for better performance. Users can leverage the aforementioned *Sessionbegin* and *Sessionend* API to cal-



(a) The heap space is divided into two areas during normal execution

(b) GC starts: partition the heap into three areas for isolated execution

(c) GC ends: re-partition the heap into two areas

(d) When the bump pointer reaches the end, it will "jump" to the start address for memory allocation

Figure 6: The heap layout in *Platinum*. The colored part stands for allocated memory.

culate the working set size for their applications during prerun, and modify the pinned area to a proper size accordingly.

During the concurrent scavenge step, GC threads will apply range checks to determine if an object falls into the collection area before accessing it, and only those in the collection area will be processed. Those range checks avoid the case where GC threads try to copy an object which mutators are modifying. Since the collection area is consecutive, range checks can be implemented with cheap comparison instructions and thus introduce little overhead. As a result, GC threads and mutators are enforced to concentrate on processing objects in different areas, which eliminates the need for synchronization and thereby mitigates the runtime overhead. For objects not processed during concurrent scavenge, they will be scanned and updated in the subsequent stop-the-world scavenge step.

Before GC ends, GC threads become inactive, and *Platinum* should reorganize the eden-space (Figure 6c). Since a part of the space has been occupied by live objects (pinned and allocation area), *Platinum* will mark it allocated. The rest of the space will still be partitioned into two areas, while the reserved one will reside adjacent to the pinned area in the last GC. The bump pointer will grow from the original allocation area and goes to the start address once it reaches the end (Figure 6d). When the allocation area is exhausted, a new GC cycle is activated.

4.5 Hardware-assisted barrier elimination

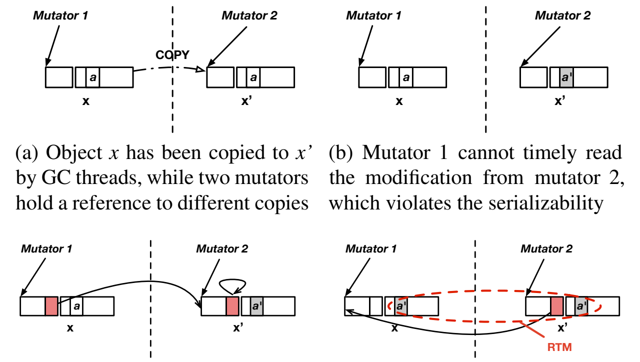
Heap partition restricts GC threads to only process objects in the collection area and thus reduces the coordination overhead. However, since mutators are running Java code, they are free to access any objects in the Java heap, including those in the collection area, which violates the *isolated execution* semantic. A traditional solution proposed by prior concurrent collectors is to leverage *barriers* to detect and correctly handle those operations. A barrier is a piece of code instrumented before specific operations for the interest of GC. For our problem, a garbage collector can adopt *write barriers*, which instrument range check operations before every write. As shown in Figure 7, for a field update operation (*y.x* = *z*), the collector should insert a range check to ensure that the address of the field (*y.x*) is not in the collection area. If the range check fails, mutators should turn to a prepared slow path (*atomic_update*) to update the field atomically to ensure correctness.

```
// barrier code start
if (in_collection_area(y.x)) {
    atomic_update(y.x, z);
}
// barrier code end
else {
    // Field update
    y.x = z;
}
```

Figure 7: An example of write barriers

As mentioned in Section 3, barriers can cause significant overhead as they are instrumented with *every* specific instruction, no matter if GC is active. For write barriers, they must be executed before every write operation, including interpreter code, JIT code, and even part of native code inside JVM. We instead provide a hardware-assisted solution atop MPK to eliminate the write barriers.

To leverage MPK, *Platinum* divides the Java heap space into two *domains*: *GC domain* and *mutator domain*. The *GC domain* only contains the collection area, while the *mutator domain* consists of the pinned area and the allocation area. When threads are initialized, they will be granted with corresponding permissions: mutators have read-write permissions to the mutator domain and read-only permissions to the GC domain; GC threads have read-write permissions for both two domains. The permissions are fixed throughout the lifecycle of a thread. On the other hand, the address ranges for those two domains are changed with the three areas dynamically. When the collector is inactive, the whole Java heap space belongs to the mutator domain so that mutators are free to access any objects. When GC starts, the collection area should be put into the GC domain. Afterward, if concurrent mutators' write operations fall into the GC domain, they will trigger page faults, and thus no software barriers are required. Thanks to MPK, *Platinum* can *automatically* detect write operations violating the isolated execution mechanism, and the control flow will be transferred to a customized handler to correctly process those operations. Although processing a page fault is more costly than executing a software barrier, the possibility of triggering a page fault in interactive services is much smaller than that for software barriers, and the amortized overhead can be reduced.



(a) Object *x* has been copied to *x'* by GC threads, while two mutators hold a reference to different copies

(b) Mutator 1 cannot timely read the modification from mutator 2, which violates the serializability

(c) *Shenandoah* exploits an indirection pointer so that accesses to the old object will be forwarded to the new one

(d) *Platinum* instead updates both two copies atomically by putting the updates into the same hardware transaction

Figure 8: The dual-copy problem and solutions

4.6 Handling violated writes

When a write operation from a mutator falls into the collection area, the customized handler will take over and temporarily request for read-write permissions for the GC domain. This request is safe as the code in the handler is totally controlled by *Platinum*. Afterward, the handler is responsible to simulate the original write operation on the mutator's behalf by modifying the corresponding object in the collection area.

However, the simulation process must be deliberately designed as GC threads are concurrently copying those objects. If an object has been copied, the old one and the copied one will co-exist in the Java heap until GC ends. Since mutators may still have references to the old object, the consistency between the two copies must be maintained. Consider the case in Figure 8a where object *x* has been copied (say *x'*). Suppose mutator 2 gains a reference to *x'* and modifies a field *a* in *x'* (Figure 8b), the modification should be visible to all mutators. However, if mutator 1 still holds a reference to *x* and reads its content, it can only get a stale value of *a*. We refer to it as the *dual-copy problem*, which breaks down the serializability of the whole program.

Prior concurrent collectors have similar problems as they also allow mutators to run in scenarios where two copies of the same object are both visible in the heap. They leverage *read barriers* to force mutators always to access the newest one. The implementation of read barriers has many variants, and one of them is to use Brook's style indirection pointers [5] as Shenandoah [13] does. This solution requires adding an extra field in the header of every Java object, which stores a pointer to the newest copy of this object. As illustrated in Figure 8c, *x* points to *x'* while x' points to itself. In this way, when mutator 1 tries to access *x*, the indirection pointer of *x* will guide it to access *x'* instead so that it can get the updated value of *a*. This solution is simple and straightforward, but it introduces considerable overhead, as analyzed in Section 3.1.

Rather than using read barriers, *Platinum* guarantees correctness by updating *both* copies in the customized page fault handler. As shown in Figure 8d, *Platinum* keeps the added field in Shenandoah to store a *back pointer* to the old copy. For the old object, since the original PSGC will store a *forwarding* pointer in its header referring to the new object, the added field is useless. In the above example, when updating *x'*, *Platinum* will locate *x* with the back pointer and update the value of *a* for both *x* and *x'*. Even though mutator 1 retains a reference to *x*, it can still fetch the updated content by directly reading *x*. This mechanism certainly doubles the write operations, but it relies on the observation that only a few write operations from mutators will happen in the collection area, so the overhead will be trivial.

Prior work like Sapphire [21] also exploits similar mechanisms, but it struggles to make the updates atomic, i.e., the updates to both copies should be visible to mutators simultaneously. Fortunately, recent hardware development has provided us with new opportunities for design. *Platinum* embraces the *Restricted Transactional Memory* (RTM) feature by Intel, which guarantees the atomicity of a piece of code by wrapping it into a hardware transaction. Since RTM requires that the working set of the transaction should be small (otherwise the abort rate will dramatically increase), we only put the update operations into the hardware transaction to construct a very small working set (less than 100 bytes). We have also prepared a fall-back handler in case the transaction fails. The handler retries the transaction in most cases, but it will try to grab a global lock if the transaction fails for many times (which happens very rarely). With RTM's help, *Platinum* can update both objects atomically and return to normal execution, which provides a strong consistency guarantee and introduces moderate overhead.

5 Evaluation

Platinum is implemented in the HotSpot JVM of OpenJDK 8u141 with about 7,500 LoCs. We leverage three various applications for evaluation:

SpecJBB2015. SpecJBB2015 is a business benchmark that simulates an online supermarket to process incoming purchasing requests. Alibaba usually exploits it as a simplified example to simulate the online production environment.

Cassandra. Cassandra is an open-sourced key-value store which is usually leveraged as a latency-sensitive application by prior work [7, 40]. We leverage YCSB as the testbed, but it is executed in a closed-loop model where a client will not send a second request until its receives the response for its last one. This model cannot reflect the fact that requests have to wait until prior ones are finished. Therefore, we have modified the execution model of YCSB to open-loop, where clients send requests in a fixed throughput regardless of the responsiveness of servers. The version of Cassandra for our evaluation is 3.11.4.

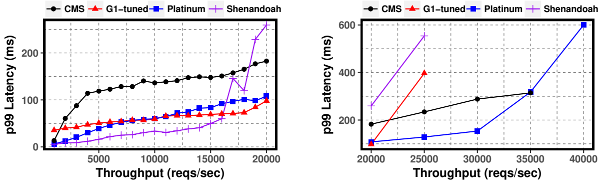
Coupon. Coupon is an online interactive service used in Alibaba, and we use it to confirm that *Platinum* actually works in real applications.

We also compare the performance of *Platinum* against other mainstream garbage collectors.

CMS. Concurrent-Mark-Sweep (CMS) is a classic partially-concurrent garbage collector. Its major GC is concurrent, and the minor GC is stop-the-world. We leave CMS untuned to show its original performance.

G1. G1GC (G1) is a highly-tunable partially-concurrent garbage collector which prioritizes latency over throughput. It is designed to replace CMS in the future versions of OpenJDK. We have manually tuned the value of *MaxGCPauseMillis* for performance consideration. Since G1 is an experimental collector in OpenJDK 8, we also try the later OpenJDK 9 for evaluation. However, OpenJDK 9 is not supported by the coupon service and Cassandra, and our evaluation on SpecJBB2015 shows similar results for those two versions. Therefore, we only report the result for OpenJDK 8.

Shenandoah. Shenandoah is a work-in-progress mostly-concurrent garbage collector. The application latency is quite low, but the introduction of *read barriers* and other concurrent phases strongly affect CPU utilization.



(a) low-throughput

(b) high-throughput

Figure 9: The p99 latency results on SpecJBB2015 for different collectors with various throughput settings.

5.1 SpecJBB2015

We use the preset mode to evaluate SpecJBB2015 atop different collectors to understand its performance under various levels of throughput. In Alibaba, different throughput settings can be used to simulate different types of workload. The experiment is running on a physical machine with dual Intel Xeon Gold 6138 CPUs (80 logical cores) and 16GB Java heap size. The number of concurrent GC threads is set as the default value (53). We tune G1 with different *MaxGCPauseMillis* values and find that it reaches the shortest per-GC pause time when the value is 50ms, so we adopt this value for evaluation (referred to as *G1-tuned*). Figure 9 shows the 99th percentile latency for *Platinum*, CMS, G1, and Shenandoah. The results in Figure 9a show that *Platinum* always performs better than CMS under moderate throughput, and the 99th percentile latency is reduced by 38.4%-79.3%. *Platinum* also achieves comparable performance against our tuned G1. The improvement in latency mainly thanks to the mostly-concurrent collection in *Platinum*.

When the throughput becomes higher (shown in Figure 9b), G1 reaches its limit at 25000 requests per second. As a mostly-concurrent collector, the latency of Shenandoah is ultra-low in low throughput but dramatically rises when the throughput is 17000 and also reaches its limit at 25000. Note that G1 and Shenandoah shares a similar concurrent marking algorithm, and the main difference in their design choices is that G1 pauses mutators during collection while Shenandoah allows concurrent execution. Therefore, G1 has better GC efficiency and performs better under high throughput, while Shenandoah induces shorter pauses and performs better under low throughput. In contrast, *Platinum* can sustain the highest throughput of all collectors, thanks to the cost-effective isolated execution model during concurrent collection.

We also measure the CPU utilization for collectors under three different QPS settings (5000, 15000, 25000) to represent low, moderate, and high throughput. As Table 2 shows, the CPU utilization of *Platinum* is only slightly higher than CMS and better than both G1 and Shenandoah under all settings. Since we did not tune CMS for application latency, it reaches reasonable CPU consumption but far worse tail latency compared against other collectors. When the throughput reaches 25000, both G1 and Shenandoah show considerable CPU consumption compared with *Platinum*, which results in the severe tail latency problem (Figure 9b), while the CPU utilization in *Platinum* is still moderate.

Name	CMS	G1	Shenandoah	Platinum
Specjbb (qps=5000)	14.57%	16.53%	17.85%	15.11%
Specjbb (qps=15000)	31.77%	37.25%	43.03%	32.79%
Specjbb (qps=25000)	48.79%	77.66%	77.80%	50.56%
Cassandra (qps=80000, RI)	11.87%	14.35%	14.07%	12.99%
Cassandra (qps=80000, WI)	12.10%	15.97%	14.93%	13.79%
Coupon (qps=4000)	38.47%	36.17%	83.05%	34.50%

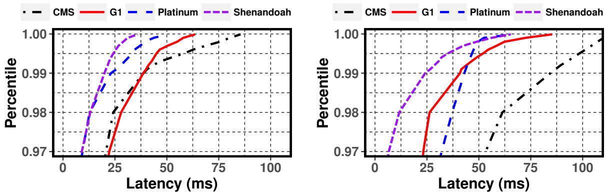
Table 2: The CPU utilization for four garbage collectors, with different applications

5.2 Cassandra

We evaluate Cassandra under the same settings as SpecJBB2015. Two different types of workload are leveraged for evaluation: (1) read-intensive workload (RI) with 76000 reads and 4000 updates per second; (2) write-intensive workload (WI) with 40000 reads and 40000 updates per second. We have also tuned G1 to achieve its best performance, and the value of *MaxGCPauseMillis* is 10ms. Figure 10 illustrates the tail latency for both scenarios with CDFs. As for the read-intensive workload, *Platinum* has comparable performance with Shenandoah, and improves the 99th percentile latency by 40.5% and 40.4% for CMS and our tuned G1 respectively. In *Platinum*, the latency for 97.2% of requests is less than 10ms, and the number is 9.5% and 3.2% larger than G1 and CMS. The improvement drops for write-intensive workload, and only 91.2% of requests finish in 10ms. This can be explained by more violated writes from mutators due to more update operations on the globally-visible data structures. Nevertheless, the p99 latency of *Platinum* is comparable with our best-tuned G1 and improved by 44.9% compared with CMS.

All collectors show moderate CPU consumption in Cassandra. It is because Cassandra is an I/O-intensive application and spends much more time on accessing its storage compared with other scenarios. Since our tuned G1 reduces the application latency by greatly shrinking the young space and increasing the accumulated GC time, it reaches the highest CPU utilization among all collectors.

Table 3 further shows GC-related statistics in 30 seconds among different collectors in the read-intensive workload. It also shows the results for three different settings in G1. The untuned CMS has the least overall time among all collectors (except for G1-100ms and G1-60ms), but its average pause time is relatively large. As for G1, when setting *MaxGCPauseMillis* from 100 to 10, the overall GC time is enlarged by 2.5X, which results in higher CPU consumption. Compared with other collectors, Platinum reaches both satisfying average GC pause time (close to Shenandoah) and overall GC time (close to G1-100ms).



(a) Read-Intensive (b) Write-Intensive

Figure 10: The CDF results for Cassandra under two different workload

GC settings	Average pause (ms)	Overall time (ms)	p99 latency (ms)
CMS	28.168	366.189	38.776
G1-10ms	16.144	1037.864	38.677
G1-60ms	38.998	775.032	62.794
G1-100ms	58.739	413.583	76.732
Shenandoah	3.97	522.499	27.700
Platinum	4.66	433.335	23.061

Table 3: GC and latency statistics for Cassandra RI

5.3 Coupon

We finally show the performance of *Platinum* on the coupon service. Since we do not have enough physical MPK-enabled machines to conduct the clustered evaluation in Section 2.2, this evaluation still exploits the single-point environment mentioned in Section 2.3 on a 96-core machine with 16GB Java heap. The throughput is set to 4000 requests per second to simulate stressful throughput in the production environment. Since we have studied the performance of G1 on the coupon service before, the value of *MaxGCPauseMillis* is set to 60ms.

Figure 11 shows the CDF generated according to the response time of requests. The results indicate that *Platinum* can mitigate the long tail problem, especially for p99 latency. Thanks to the cost-efficient garbage collection, the p99 latency in *Platinum* is improved by 66.8% and 23.5% for CMS and G1. Since the real-world workload also contains requests whose response time is greatly extended by lags in other remote services, *Platinum* cannot help to improve them much and result in slightly better p999 latency against other collectors. The application latency for Shenandoah is very large (for example, the 99th percentile latency is 3.6s), which is out of range in Figure 11. Compared with statistics in Table 1, the average pause time and GC count in *Platinum* is 7.247ms and 162 respectively.

As for CPU utilization, *Platinum* consumes 34.5% of overall CPU resources, which is the smallest among all collectors. The CPU consumption is 1.67% and 3.97% smaller than G1 and CMS. As for Shenandoah, the CPU is close to saturated (83.05%), which can explain why application latency is quite large. To conclude, the evaluation results on all three applications confirm that *Platinum* finds a sweet spot between low application latency and moderate CPU utilization.

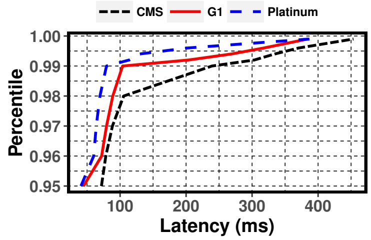


Figure 11: The CDF results for the coupon service

5.4 Breakdown analysis

Varying the pinned area size. Table 4 shows the runtime statistics of the Cassandra-WI workload with different sizes of the pinned area. When enlarging the pinned area, the average number of page faults for each GC cycle decreases, and the tail latency can be slightly improved. However, since the pinned area will only be reclaimed in the next GC cycle, enlarging its size will reduce the available memory in the eden space and increase the overall GC time. Therefore, users can tune the size of the pinned area to reduce the tail latency according to the application behavior.

Area size	Overall GC time (ms)	Avg. page faults	p99 latency (ms)
1/128	1007.327	13738	43.038
1/32	1106.951	12137	41.391
1/16	1145.982	11879	43.773
1/8	1206.859	10237	39.619

Table 4: GC-related statistics for Cassandra WI

GC Performance breakdown. This experiment breaks the accumulated GC time of Cassandra-WI into phases. As shown in Figure 12, mutators are allowed to run concurrently with GC threads most of the time (78.3%). Since the initial marking phase only scans the thread stacks, it lasts shortly and only takes up 1.1%. Meanwhile, the STW scavenge phase accounts for 15.6% to modify the stale references in the pinned area and the collection area.

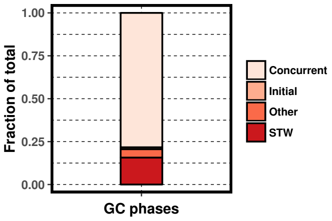


Figure 12: Phase-level breakdown for *Platinum*

The performance on Spark. We also evaluate the performance of *Platinum* on Spark, with its built-in *PageRank* application. Our evaluation shows that the execution time with *Platinum* is 7.93% longer compared with G1. Since the memory and GC behavior in Spark is much different from interactive services, it induces more page faults and larger stop-the-world pauses, which is the main reason for performance slowdown.

6 Related Work

Reducing pause time. STW pauses introduced by garbage collections have been studied for years. For STW garbage collectors, some work aims to reduce the pause time by sufficiently leveraging the computing resources. Gidra et al. [15, 16, 17] study the performance of PSGC in NUMA machines and provide NUMA-aware optimizations. Suo et al. [40] and Qian et al. [37] refine the work-stealing algorithm to offer a more scalable minor GC algorithm. Another line of work focuses on designing new concurrent collectors to co-run mutators with GC threads to reduce the pauses. In the OpenJDK HotSpot JVM, the latency-aware G1GC [11] has recently become the default collector, and two mostly-concurrent collectors, ZGC [33] and Shenandoah [13], are also attractive. Stopless [36] provides real-time GC support while preserving lock-freedom and fragmentation control. Österlund et al. [34] improve the pause time of G1GC with a non-blocking handshake between threads. The design of *Platinum* learns from those concurrent collectors to propose a CPU-efficient solution.

The intensive usage of language runtime in the big data area has stimulated studies on building big-data-friendly garbage collector with low pauses. Nguyen et al. [31, 32] put the data objects generated by the big data systems into segregated spaces where objects are managed with separated GC algorithms. Gog et al. [18] provide a similar region-based algorithm but mainly for the CLR runtime. Bruno et al. [6, 7, 8] divide the heap into many generations and pre-tenure objects that are believed to live long through runtime analysis. *Platinum* is built for reducing pauses for another kind of scenario: latency-sensitive, session-based interactive services.

Hardware-assisted GC. Hardware features also affect the design choices of garbage collectors. Prior work has studied on improving the performance of GC with primitives available in commodity hardware. Belay et al. [3] leverage virtualization technology to escalate Java runtime to the non-root ring 0 mode and accelerate GC with VM page management. Ugawa et al. [42] enhance the original Sapphire garbage collector with the RTM feature, and Ritson et al. [39] explore the usage of RTM in various collectors. Wu et al. [45] extend the area of garbage collector from normal DRAM to non-volatile memory (NVM) and studies crash consistency issues during GC. *Platinum* leverages RTM and MPK features to build a new concurrent garbage collector.

Except for commodity hardware, there is also a trend to build customized hardware, or GC accelerators, for various considerations. Azul Systems has built a customized system including CPU, chip, board, and OS to run garbage collected JVMs efficiently. Their GC algorithm is also largely modified to leverage those customized features [10, 23, 41]. Mass et al. [26] build a hardware GC accelerator to achieves higher GC throughput and lower power consumption.

Runtime optimization in distributed environments. Distributed applications are running atop multiple runtimes on different machines. Maas et al. [25, 27] show that GC in a single JVM can have a magnified effect on the whole applications and proposes policies to orchestrate GC among different JVMs. Lion et al. [24] find frequent JVM re-start in task-based workload and provides a JVM pool to skip the time-consuming warm-up phase. Nguyen et al. [30] optimize the communication between JVMs by providing an efficient serialization protocol, while Navasca et al. [29] allow Java threads to directly operate on the serialized byte streams with compiler techniques. Wang et al. [44] propose to dynamically change the memory limits of JVMs to fit the cloud environment. Both Fang et al. [12] and Călin et al. [22] leverage efficient spilling to reduce the memory footprint of large data-parallel applications. *Platinum* also quantifies the effect of GC in a distributed cloud environment and proposes a hardware-assisted algorithm to optimize the GC performance.

7 Conclusion

Latency and CPU efficiency are both essential for interactive services. Unfortunately, traditional garbage collectors cannot achieve both goals at the same time. This paper provides *Platinum*, a collector allowing concurrent but isolated execution of mutators and GC threads, with hardware assistance. The evaluation shows that *Platinum* significantly reduces the tail latency while preserving moderate CPU utilization compared with prior concurrent garbage collectors.

8 Acknowledgement

We sincerely thank our shepherd Gilles Muller and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61672345, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Haibo Chen is the corresponding author.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2009.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *ACM SIGPLAN Notices*, volume 23, pages 11–20. ACM, 1988.

[3] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

[4] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI*, volume 91, pages 157–164. Citeseer, 1991.

[5] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262. ACM, 1984.

[6] R. Bruno and P. Ferreira. Polm2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 147–160. ACM, 2017.

[7] R. Bruno, L. P. Oliveira, and P. Ferreira. Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications. *ACM SIGPLAN Notices*, 52(9):2–13, 2017.

[8] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 28. ACM, 2019.

[9] A. Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, page 13, 2014.

[10] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56. ACM, 2005.

[11] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[12] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 394–409. ACM, 2015.

[13] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 13. ACM, 2016.

[14] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. {IMIX}: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018.

[15] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 7. ACM, 2011.

[16] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *ACM SIGPLAN Notices*, volume 48, pages 229–240. ACM, 2013.

[17] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: a garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[18] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[19] S. Han, S. Lee, S. S. Hahn, and J. Kim. Syncgc: A synchronized garbage collection technique for reducing tail latency in cassandra. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, page 20. ACM, 2018.

[20] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[21] R. L. Hudson and J. E. B. Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57. ACM, 2001.

[22] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 97–109, 2017.

[23] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The collie: a wait-free compacting collector. In *ACM SIGPLAN Notices*, volume 47, pages 85–96. ACM, 2012.

[24] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 383–400. USENIX Association, 2016.

[25] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *ACM SIGOPS Operating Systems Review*, 50(2):457–471, 2016.

[26] M. Maas, K. Asanović, and J. Kubiatowicz. A hardware accelerator for tracing garbage collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 138–151. IEEE Press, 2018.

[27] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[28] S. Microystems. Memory management in the java hotspot™ virtual machine, 2006.

[29] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 538–553. ACM, 2019.

[30] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ACM SIGPLAN Notices*, volume 53, pages 56–69. ACM, 2018.

[31] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[32] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*, 2015.

[33] OpenJDK. ZGC - The Z Garbage Collector. https://openjdk.java.net/projects/zgc/, 2019.

[34] E. Österlund and W. Löwe. Block-free concurrent gc: stack scanning and copying. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 1–12. ACM, 2016.

[35] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[36] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pages 159–172, 2007.

[37] J. Qian, W. Srisa-an, D. Li, H. Jiang, S. Seth, and Y. Yang. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for java vm. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 170–181. ACM, 2015.

[38] J. Qian, W. Srisa-An, S. Seth, H. Jiang, D. Li, and P. Yi. Exploiting fifo scheduler to improve parallel garbage collection performance. *ACM SIGPLAN Notices*, 51(7):109–121, 2016.

[39] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with haswell hardware transactional memory. In *ACM SIGPLAN Notices*, volume 49, pages 105–115. ACM, 2014.

[40] K. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*, page 35. ACM, 2018.

[41] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46(11):79–88, 2011.

[42] T. Ugawa, C. G. Ritson, and R. E. Jones. Transactional sapphire: Lessons in high-performance, on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(4):15, 2018.

[43] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1221–1238, 2019.

[44] J. Wang and M. Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 745–758, Santa Clara, CA, 2017. USENIX Association.

[45] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 70–83. ACM, 2018.

PinK: High-speed In-storage Key-value Store with Bounded Tails

Junsu Im, Jinwook Bae, Chanwoo Chung*, Arvind* and Sungjin Lee

DGIST

Massachusetts Institute of Technology

Abstract

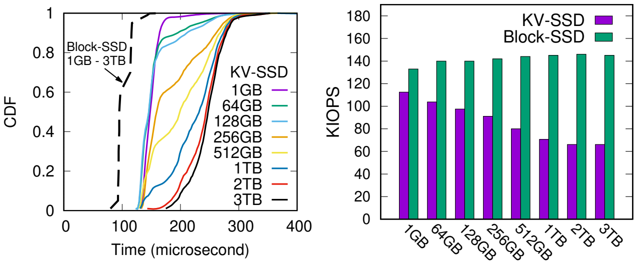
Key-value store based on a log-structured merge-tree (LSM-tree) is preferable to hash-based KV store because an LSM-tree can support a wider variety of operations and show better performance, especially for writes. However, LSM-tree is difficult to implement in the resource constrained environment of a key-value SSD (KV-SSD) and consequently, KV-SSDs typically use hash-based schemes. We present *PinK*, a design and implementation of an LSM-tree-based KV-SSD, which compared to a hash-based KV-SSD, reduces 99^{th} percentile tail latency by 73%, improves average read latency by 42% and shows 37% higher throughput. The key idea in improving the performance of an LSM-tree in a resource constrained environment is to avoid the use of Bloom filters and instead, use a small amount of DRAM to *keep/pin* the top levels of the LSM-tree.

1 Introduction

Offloading the key-value (KV) functionality onto a storage device has received a lot of attention recently from both academia and industry [11, 21, 24, 32, 49]. A representative device in this class is Samsung's key-value SSD (KV-SSD) [24], which directly serves KV requests. By offloading most commonly used operations of KV databases (e.g., RocksDB [17]), KV-SSDs not only improve I/O latency and throughput of KV clients, but also reduce the CPU and DRAM resource requirements on the host-side.

The idea of KV-SSD is promising but the current proposals and devices often provide inconsistent tail latency and throughput. This is because most of KV-SSDs are based on hash [16, 21, 24, 32, 46, 49], which is attractive because it is rather simple to implement but has some inherent limitations. A hash-based KV-SSD maintains a hash table in the controller DRAM, each entry of which typically contains a key (or the signature of a key) and a pointer to the corresponding KV pair in the flash. The hash table is used to quickly index key-value pairs by simple table lookups. However, when the DRAM size is not large enough to accommodate all the hash table entries, parts of the hash table must be stored in flash. This inevitably involves expensive flash accesses and complex hash table management when accessing entries that are not in memory. Even worse, if a hash collision occurs, multiple flash accesses are required, resulting in long and unpredictable tail latency and drop in throughput.

To understand the behavior of hash-based KV-SSDs, we conducted a set of experiments on a 4TB KV-SSD prototype (KV-PM983 [40]). We created KV pools ranging in size from 1GB to 3TB, and chose the average key and value sizes to be 32B and 1KB, respectively [5]. Thus, a 3TB KV pool would hold 3 billion KV pairs. We ran random `GET()` requests on these KV pools for 10 minutes using KVBench [38]. No GC occurred during our experiments.



(a) CDF of read latency (b) Read throughput

Figure 1: Performance comparison of KV-SSD & Block-SSD depending on the total amount of data stored (1GB∼3TB)

Figure 1 shows that the KV-SSD suffered from inconsistent read latency, and its throughput dropped as the number of objects stored increased. The average read latency increased from 149.49 μs (1GB pool) to 245.31 μs (3TB pool). We also observed long tail latency: for the 99.99^{th} percentile, the tail latency increased from 323 μs to 1020 μs. Even worse, the read throughput dropped to 64 KIOPS from 112 KIOPS. Although we did not have access to any of the internal details of the KV SSD design (*e.g.,* the hash function), it is easy to conclude that the performance and tail latency get worse in a hash-based implementation as the total number of stored KV pairs increases. This hypothesis was further supported by another experiment, where we used the same setup to run FIO [6] on a 4TB Block-SSD [41], which loads its FTL table in DRAM for 4KB page mapping. FIO exhibited stable latency and throughput, regardless of the amount of data stored. Such severe performance variability and unpredictable I/O behaviors make KV-SSDs less attractive than normal SSDs.

An alternative to hash is *log-structured merge tree* (LSM-tree) [34]. The tail latency in such a system is bounded by the number of levels in the tree. Since an LSM-tree indices KV pairs in a multi-level *sorted* tree, it might also require a much smaller DRAM for indexing KV pairs. LSM-tree also supports range-queries and scans efficiently, without any extra bookkeeping or support from KV clients [24]. Our experiments, however, revealed that a conventional implementation of LSM-tree on an SSD controller failed to deliver the promised benefits. In fact, it showed worse performance than hash in some cases.

The first problem we discovered was the tail latency. Most LSM-tree implementations use Bloom filters to skip lookups in a tree level to improve *average* read latency. Owing to the probabilistic nature of Bloom filters, however, one cannot ensure the *worst-case* read latency; indeed, we observed long tails as in hash-based KV-SSD. The second problem was high write-amplification. Even if we use key-value separation like Wisckey [30], compaction, an essential task of LSM-tree to sort KV indices and balance its indexing trees, involves many extra storage accesses. Moreover, this LSM-tree compaction cost exacerbates the FTL's garbage collection (GC) cost. The third problem was that rebuilding Bloom filters and sorting KV pairs for compaction requires lots of CPU cycles, which overburden embedded-class microprocessors found in SSD controllers. This lack of processor performance deteriorates the I/O performance dramatically.

In this paper, we propose an LSM-tree-based in-storage key-value engine, called *PinK*, which overcomes all the problems mentioned above. The novelty of PinK design stems from four specific techniques it uses. At the heart of PinK is *level pinning*. Instead of keeping probabilistic Bloom filters in DRAM, PinK pins exact key-value indices of the top levels of the tree to DRAM. This removes unnecessary flash lookups on the pinned levels in a deterministic manner, thereby enabling us to provide predictable read latency with bounded tails. Elimination of Bloom filters also reduces the resource requirement for computing them. Second, the level pinning helps us reduce flash I/Os caused by compaction. Since KV indices are kept in DRAM, PinK can sort them in DRAM without any I/Os. The pinned indices are protected by built-in capacitors, so flushing out up-to-date indices to flash is not necessary. (This idea is feasible only for small amount of DRAM). Third, we discovered that the majority of GC I/Os are induced by updating indices of LSM-tree. By delaying index updates until the compaction phase, PinK reduces GC I/Os greatly. Finally, by adding hardware comparators in between the SSD controller and NAND chips, and performing KV sorting while reading KV pairs, PinK completely eliminates CPU costs for compaction.

We have implemented PinK on MIT's FPGA-based SSD platform [22], and used the LSM-tree implementation of LightStore [11] as our starting point, because its source code is publicly available. Using YCSB [13] benchmarks, we have shown that PinK outperforms existing KV-SSD designs in several aspects. Compared to a hash-based KV-SSD, PinK reduces 99^{th} percentile tail latency by 73%, improves average read latency by 42% and shows 37% higher throughput. Furthermore, compared to LightStore, PinK reduces 99^{th} percentile tail latency by 22%, improves average read latency by 22% and shows 44% higher throughput.

Paper Organization: In Section 2, we explain background closely related to this study. Section 3 analyzes the performance of the LSM-tree algorithm in KV-SSD. Section 4 presents an overall design of PinK, along with optimization techniques. Section 5 presents experimental results. We conclude in Section 6.

2 Background

2.1 NAND Flash-based SSD

A conventional Block-SSD is designed to support the standard block I/O interface. It exposes a linear array of 4KB logical blocks which are accessed by block I/O primitives (*e.g.,* `READ` and `WRITE`). A flash translation layer (FTL) in the SSD firmware is responsible for providing the block I/O interface [3]. To hide the out-of-place update nature, the FTL writes incoming data to free flash pages in an append-only manner. To redirect 4 KB logical blocks to free pages, the FTL maintains a mapping table indexed by logical block address (LBA), and each entry points to the corresponding flash page. The mapping table is kept in the controller DRAM and its size is approximately 0.1% of the SSD capacity [39, 43]. For example, for a 4TB SSD, 4GB DRAM is required. A mapping table has to be persistent (non-volatile) and is protected by built-in capacitors to guard against sudden power failures [7]. Similar to other log-structured systems [36], the FTL has to perform garbage collection (GC) to reclaim free space.

2.2 KV-SSD

A KV-SSD is a new type of SSDs [24, 45] which provides the key-value interface. KV-SSDs look like a container of key-value objects, where each object is labeled by a unique key and contains an associated value (*i.e.,* data). In contrast to a block addressed SSD, both the key and the associated value are of variable sizes. A key can be as long as 255 bytes [45] or even be a character string, and a value can be as big as 2MB [45]. In addition to `GET()` and `SET()`, the basic operations to access KV objects, KV-SSDs support a rich set of operations like iterations, range queries, and transactions [24, 25]. A more detailed description can be found in SNIA's KV-SSD specification [45].

Making SSDs support the KV interface requires a redesign of the FTL because the existing table-based translation is not suitable for managing KV objects. A variety of KV-SSD designs have been proposed both in academia (*e.g.,* NVMKV [32], KAML [21], and BlueCache [49]) and industry (*e.g.,* Samsung's KV-SSD prototype [24, 40]). All these KV-SSDs are based on hash-based data structure, which we discuss next.

2.3 Hash-based KV-SSD

A hash-based KV-SSD maintains a hash table with many buckets in DRAM, where each bucket holds metadata (*i.e.,* a key and a pointer) for a specific KV object in flash [16, 21, 32, 49]. A primary design issue of hash-based KV-SSD is the management of a huge hash table requiring large amounts of DRAM. Suppose that the SSD capacity is 4 TB and the key and value sizes are on average 32B and 1KB, respectively [5]. If the number of buckets is 2^{32} ($= 2^{42}/2^{10}$) and the bucket size is 36B (32B for a key and 4B for a pointer), 144GB of DRAM is required to hold the complete hash table. If KV-SSDs have large enough DRAM to hold the entire hash table, in addition to the $O(1)$ time complexity for calculating an index, a KV access only takes $O(1)$ flash access to read/write the KV pair [44]. However, as mentioned previously, SSDs do not have as much DRAM.

To reduce DRAM usage, some use signatures [9, 16, 26, 46, 49]. Instead of an exact key, a short signature of the key is kept in the bucket. The exact key and its value are stored in the flash. Using signatures reduces the hash table size greatly – if a 16-bit signature is used, 24GB of DRAM is required. But it causes *signature collision* which happens when different keys have the same signature. 24GB DRAM is still huge for an SSD. The DRAM size can be further reduced by keeping only popular buckets in a fixed-size DRAM (*e.g.,* 4GB) while storing the rest in the flash [18]. This, however, causes extra flash reads. If a designated bucket is not available in DRAM (*i.e., hash table miss*), we have to fetch the bucket from the flash to find the location of a desired KV object. Consequently, the table miss increases flash read costs from $O(1)$ to $O(1+\alpha)$ where α is a miss ratio. Even worse, signature collisions add an unpredictable number of flash reads until the collision resolves, resulting in unbounded read tail latency in the worst case. As shown in Figure 1, this instability of the hash-based KV-SSD deteriorates as the hash table grows.

This inconsistent performance may be due to inefficient collision resolution policies. There are advanced hash strategies, such as Cuckoo [35] and Hopscotch [19, 26], which provide constant worst-case lookups and may avoid the tail latency. But this benefit comes at the cost of degraded write speed and/or frequent rehashing. Hash algorithms also cannot efficiently support range and scan operations [24].

2.4 LSM-Tree-based KV-SSD

An LSM-tree is another data structure that is used widely to implement persistent key-value stores. It is usually implemented purely in host software and can support a wider set of KV operations (*e.g.,* RocksDB [17] and Cassandra [27]). It is also used in big all-flash array (AFA) systems such as Purity [12]. Because of its increasing popularity across a variety of systems, many LSM-tree variants have been proposed [4, 23, 47]. Recently LSM-tree has also been used in some implementations of KV-SSDs like LightStore [11] and iLSM-SSD [28].

LSM-tree is a hierarchical structure that consists of multiple trees, each called a *level*. Each level is sorted and behaves as a write buffer for the next level, which has a larger size. Since LSM-tree keeps only the highest level in DRAM, its memory requirement is much smaller than hash. Also, a KV access in LSM-tree requires at most $O(h-1)$ flash accesses owing to its sorted nature, where h is the number of levels, and thus the worst-case latency of LSM-tree is bounded. Many LSM-tree implementations use Bloom filters to improve the average read cost to $O(1)$ flash access [14].

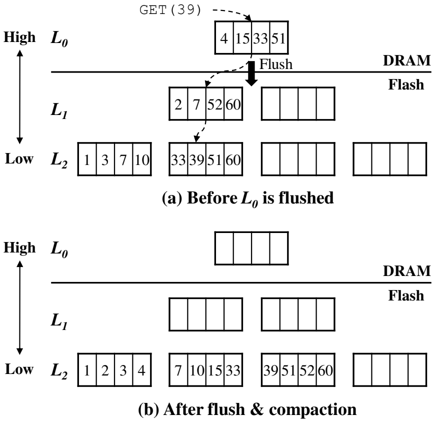
However, LSM-tree-based KV-SSDs have suffered from a lack of CPU power for the compaction process, which is required to keep LSM-tree balanced. In addition, Bloom filters, which is used to skip some levels probabilistically, cannot improve the read tail latency. The design of PinK was motivated by these and some additional inefficiencies in running the conventional LSM-tree on resource-constrained KV-SSDs; we discuss these inefficiencies in Section 3.

2.5 Hash vs LSM-Tree

NAND flash scales faster than DRAM. According to [20], the capacity of NAND flash has increased 1.43 times per year, while that of DRAM has increased 1.13 times. This requires us to take into account a DRAM scalability issue in choosing algorithms for KV-SSDs. Assuming the same trend, the hash suffers from more degradation in read performance since the table miss rate becomes higher as NAND flash scales further. The LSM-tree also experiences degradation since it uses fewer bits per KV pair for Bloom filters. However, the read performance of the LSM-tree is more scalable due to the space-efficient structure of Bloom filters. Monkey has almost the same read performance when the ratio between the size of DRAM and total data set decreases from 0.16% to 0.02% (see Figure 11(a) in [14]). Conversely, the hash suffers from notable performance degradation when the DRAM size does not grow relative to the data set as shown in Figure 1. The LSM-tree also exhibits lower update costs than the hash when the entire indices do not fit in DRAM. In the hash, indices in the map have to be updated in place, which in turn involves expensive read-modify-writes in the flash. On the other hand, the LSM-tree shows cheaper update costs since it appends new or updated entries to the flash thanks to its leveled structure. As a result, the LSM-tree offers better write performance when DRAM becomes less sufficient.

3 Challenges in implementing LSM-tree in a KV-SSD

In this section, we analyze the performance and present key technical challenges when an LSM-tree is implemented in a resource constrained environment of an SSD controller. We have used LightStore [11] as the baseline for an LSM-tree-based KV-SSD which separates keys and values to speed up writes and uses Bloom filters to speed up reads. We expect iLSM-SSD [28] to exhibit similar read and write performance as LightStore because it follows same concepts.



(a) Before L_0 is flushed

(b) After flush & compaction

Figure 2: LSM-tree organization ($h = 3$, $T = 2$). A rectangle represents a KV object and the number inside is the key.

3.1 LSM-Tree Structure

The LSM-tree maintains multiple levels of sorted KV indices, L_0, L_1, ..., and L_{h-1}, where h is the height of an LSM-tree (see Figure 2). The level 0, L_0, is kept in DRAM as a write buffer, whereas the rest are stored in persistent media (*e.g.,* flash). In the LSM-tree, the levels are organized so that a lower level is T times larger (*i.e.,* the size factor T) than a higher one. Each level is divided into fixed-size runs, where the size of each run is usually the same as that of L_0.

The LSM-tree has two unique properties: **#1.** for each level, KV objects are unique and kept sorted by their keys; and **#2.** the key range of one level may overlap the key range of other levels due to overwrites (see Figure 2).

When a SET() request comes, a KV object is first buffered in L_0. Once L_0 becomes full, buffered KV objects are flushed out to L_1. All the objects in L_0 are written to L_1 in an append-only manner. Similarly, once L_i becomes full, its KV objects are evicted to L_{i+1}. Since the key ranges of adjacent levels may overlap, flushing out KV objects from a higher level to a lower level has to be done in a manner not to violate Property #1. Therefore, the LSM-tree algorithm performs a process called *compaction* while flushing KV objects to a lower level. Compaction reads objects from two adjacent levels, sorts them in the memory, and writes the sorted objects to next lower level as shown in Figure 2(b). Compaction incurs a huge I/O overhead. This overhead can be mitigated by separating keys from values and by avoiding moving values which are not affected by compaction (see Wisckey [30]).

The LSM-tree maintains an in-memory data structure that points to runs of levels in the flash. Each run contains a header that holds the locations of KV objects (KV indices) in the flash. Searching for a key at a specific level is fast. Once a header is read from the flash, the location of a desired KV object can be quickly found since they are sorted by key. However, finding the desired key in the entire tree requires looking in multiple levels because key ranges at different levels may overlap (Property #2). In the worst case, all levels have to be searched as shown by GET(39) in Figure 2(a). The number of the worst-case flash lookups is $O(h-1)$ (Note: L_0 is excluded since it stays in DRAM). Bloom filters are often used to avoid useless lookups on levels that do not have desired keys [14, 15]. Usually, each level or run in the tree has its own filter.

3.2 Performance Analysis

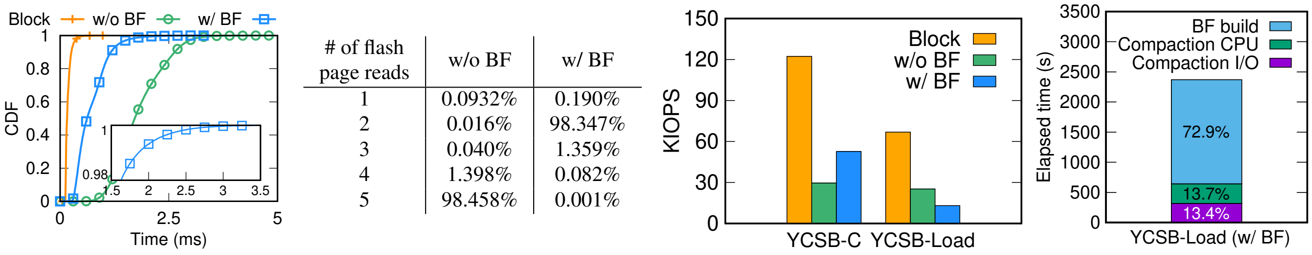
Our PinK implementation uses the same FPGA-based hardware platform as LightStore [11], which has quad-core ARM Cortex A53 running at 1.2GHz and 4GB DRAM. This controller specification is similar to those of latest SSDs with in-storage computation capability [33, 37]. PinK is equipped with a 256GB NAND flash card which provides 1.1 GB/s read and 600 MB/s write throughputs, respectively, and offers 122,349 IOPS for 4KB reads and 66,843 IOPS for 4KB writes, respectively.

To understand the weaknesses of the conventional LSM-tree implementations, we first improved the Bloom filter implementation in LightStore [11] by replacing the original one with Monkey [14]. We leveraged AArch64 SIMD instructions in implementing Monkey. Key-value separation [30] was employed by default.

For fast evaluation, we reduce the SSD capacity to 64GB. The number of levels in the tree is set to 5 ($h = 5$) with a size factor of 23. We assume that 64 MB of DRAM (0.1% of 64GB) is available and it is used to keep Bloom filters for levels. We either enable or disable Bloom filters (Monkey) to understand its impact on performance. To characterize basic performance, we run two extreme workloads, YCSB-Load (100% writes) and YCSB-C (100% reads). Average key and value sizes are 32B and 1KB. We first run YCSB-Load with 44 million (44GB) uniformly random KV-pairs, and then run YCSB-C with 10 million Zipfian requests. Results with other workloads can be found in §5.

To understand the impact of the LSM-tree algorithm, we compare the performance of the LSM-tree KV-SSD with that of a Block-SSD implemented on the same platform. The Block-SSD employs a page-level FTL whose flat mapping table, indexed by LBA, can be loaded entirely on DRAM. A physical page mapped to a logical block can be found with only one memory reference.

Figure 3(a) shows the CDF of the read latency of YCSB-C. Without Bloom filters, the LSM-tree KV-SSD shows long read latency over the Block-SSD. Figure 3(b) summarizes the number of flash page reads to service a GET() request. If GET() is directly served by L_0 (*i.e.,* a write buffer), a page read is not necessary. Otherwise, the LSM-tree looks up lower levels to fetch KV indices from the flash. The majority (98.4%) of GET() requests touch up to the last level (L_4), issuing four page reads. This is because almost all the KV pairs (95%) are stored on L_4. When Bloom filters are enabled, it offers better read latency, but is affected from long tails. With Bloom filter,



# of flash page reads	w/o BF	w/ BF
1	0.0932%	0.190%
2	0.016%	98.347%
3	0.040%	1.359%
4	1.398%	0.082%
5	98.458%	0.001%

(a) Read latency (YCSB-C) (b) Flash page read counts (YCSB-C) (c) I/O throughput (d) Compaction time breakdown

Figure 3: Experimental results of the conventional implementation of LSM-tree on an SSD controller

on average, one flash lookup is required for retrieving a KV object as in Figure 3(b). Owing to its probabilistic nature, however, 1.4% of the total `GET()`s still require more than one flash lookup, which are large enough to cause long tails (see the zoom-in figure in Figure 3(a)).

Figure 3(c) illustrates the I/O throughput. The read throughput of the LSM-tree with Bloom filter in YCSB-C is about half of the throughput that the Block-SSD provides. This is expected because Monkey requires two flash reads, on average, for retrieving KV indices to serve `GET()`.

As we can see in Figure 3(c), in YCSB-Load, we observe serious drops in the write throughput, compared to the Block-SSD. Even with Wisckey, compaction I/Os account for 75.5% of the total I/Os (both reads and writes). While not included in Figure 3(c), I/Os for GC also badly affect the write throughput. According to our analysis (see §5.2), the write amplification factor (WAF), which is 2.52 when only compaction I/Os occur, increases to 5.02 once GC starts to trigger. We find that moving valid pages for GC involves cascade updates of KV indices maintained by the LSM-tree.

The high CPU overheads of the LSM-tree also slow down the write throughput. Due to slow speed of ARM CPUs, sorting KV pairs for compaction, which involves string comparisons, becomes a bottleneck. As shown in Figure 3(d), it takes almost the same time as performing compaction I/Os. The CPU time does not include the Bloom filter reconstruction time which will be discussed soon. The compaction overhead has been addressed by KVell [29], but it requires a huge DRAM to hold all indices, which is not available in KV-SSDs.

The cost of rebuilding Bloom filters is also high. Bloom filters should be rebuilt for newly created levels after compaction, which requires expensive hash computations and lots of memory accesses. In our experiment, a hash computation is accelerated by SIMD instructions, but its negative impact is still huge. The rebuilding overhead for Bloom filters can be optimized as was done in [8]. Even if we assume the reconstruction time improves significantly, say 8X-11X, as in [8], the Bloom filter reconstruction still takes 20-25% of total compaction time. Note that it is unclear whether such huge improvement is achievable in ARM-based SSD controllers. Be advised that, our LSM-tree is carefully designed so that I/Os and computation are maximally overlapped. However, this cannot completely hide high computation costs.

The problems we have observed can be summarized as follows: **#1.** LSM-tree exhibits higher average-latency because of multi-level search, and also exhibit unpredictable tail latency because of Bloom filters; **#2.** Bloom filters require lots of computational power to reconstruct. They have to be reconstructed after each compaction; **#3.** Level compaction (excluding Bloom filter reconstruction) also requires a lot of computation and I/O bandwidth; **#4.** Compaction I/Os may trigger GC which in turn generates more I/Os, resulting in high write amplification.

4 Design of PinK

Bloom filters are used to reduce the average read latency. Another way of reducing the read latency would be to keep popular KV indices in DRAM. LSM-tree by nature keeps the recently written indices in the top levels. In PinK, we eliminate the Bloom filters and mitigate the increased read latency by pinning top-K levels (§4.2 and §4.3). We will show that level-pinning requires only a small amount of DRAM. Tail latency is already bounded to the height of the tree. Another benefit of level-pinning is that it eliminates the flash I/Os required for compaction of two levels which are already pinned in DRAM. The throughput can be further improved by using hardware accelerators that performs compaction for pinned and flash-resident levels (§4.4). Finally, to alleviate the GC costs associated with compaction, we delay GC by putting updated KV indices in L_0 (§4.5). This reduces the write amplification which affects lifetime of SSDs.

4.1 Overall Architecture

PinK supports variable-sized keys (16B∼128B) and values (1KB∼2MB), along with a rich set of KV operations (*i.e.,* `GET()`, `SET()`, `DELETE()`, `SCAN()`, and `ITERATOR()`), except for a few features like namespaces. Like KV-SSDs, PinK is able to guarantee durability and atomicity of KV operations [7, 24, 42]. While the lack of space does not permit us to describe the details of all the operations, we focus on explaining key data structures and operations which are different from the conventional LSM-tree-based KVSs.

Data Structures. Figure 4 illustrates four types of data structures of PinK: a *skiplist* and *level lists*, which all reside in

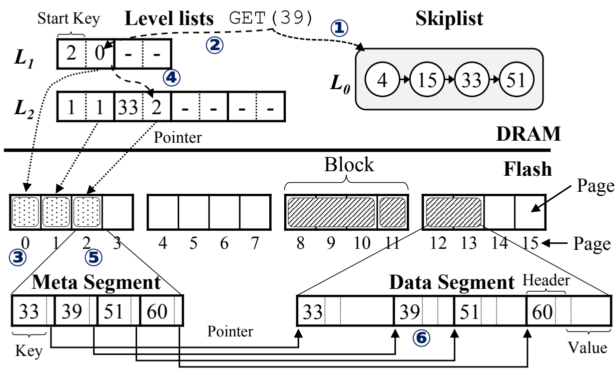


Figure 4: An overall architecture of PinK with its key data structures in DRAM and flash. The tree hierarchy and KV objects are identical to those of Figure 2(a). Given `GET(39)`, ① PinK first looks up the skiplist (L_0). Since a matched one is not found, ② it goes down to L_1, ③ reading a meta segment from the page 0. It does not have a desired key, so ④ PinK visits L_2 and reads ⑤ the page 2 to get a meta segment. Finally, ⑥ it can find the location of the value for the key '39'. Three flash reads are required to serve `GET(39)`.

DRAM, and *meta segments* and *data segments*, which all reside in flash. Overall, the design of PinK is not much different from the LSM-tree-based KVS combined with Wisckey [30], but it is optimized to maintain compact data structures in the controller DRAM for better performance in storage devices. Also, the headers of each data structure are designed to be handled easily by HW accelerators. PinK directly deals with NAND chips to perform indexing, GC, and wear-leveling obviating any need for a costly FTL found in most SSDs.

A *skiplist* corresponds to L_0 in the LSM-tree algorithm and works like a write buffer which buffers incoming KV objects temporarily. The size of L_0 is configured to be large enough (*e.g.,* 8MB$\sim$64MB) to fully utilize the parallelism of multiple NAND channels when KV objects are flushed out to the flash. Each skiplist entry has four fields: <key size, key, value size, value>, and all the entries are sorted by key.

Once the skiplist becomes full, buffered objects are materialized to L_1 as the forms of *meta segments* and *data segments*. In L_1 (and all the lower levels), keys and values are separated into meta and data segments, respectively. A meta segment contains keys and pointers to its associated values in data segments. In addition to values, a data segment stores keys and their sizes to support GC (see §4.5). The size of a meta segment is fixed to a flash page size (*e.g.,* 8KB $\sim$ 16KB), but a data segment can be of any size – it is like a huge log containing KV objects pointed to by meta segments.

Since meta segments are referenced by the software to look for a KV object and by the hardware accelerators for compaction, they are organized to be manipulated by both of them. A meta segment is composed of an array of <key, pointer> pairs sorted by key, plus a header. A pointer is a 4B integer, but a key size varies from 16B to 128B. To quickly find a variable-size key using binary search, a meta segment header maintains the start locations (2B each) of <key, pointer> pairs. If a meta segment is 16KB, it contains up to 1024 <key, pointer> pairs where at most 2KB is used as a header. For HW accelerators, a header and <key, pointer> pairs are aligned to 16B for simple implementation. We discuss this in §4.4 in detail.

PinK maintains another in-memory data structure, *level lists*, which keep track of meta segments at every level in the flash. If the tree has five levels (*i.e., $h = 5$*), there are four level lists except for L_0. Each level list is organized as an array of pairs of fixed-sized pointers (4B each, 8B total); the first one points to the physical location of a meta segment in the flash; the second one points to a start key of that meta segment. Note that start keys of meta segments are stored separately in DRAM to support variable-sized keys (16-128B). This facilitates us to implement binary search to find a desired meta segment in a level list.

Two in-memory data structures, L_0 and the level lists, are protected by capacitors. This provides enough time for PinK to safely flush out them to the flash in the event of power failures or when a system is turned off. PinK also does not need to use a write-ahead log (WAL) to provide atomicity and durability of data.

Data Structure Size. Compared to the hash, PinK requires much smaller DRAM for indexing KV objects. Assume that an SSD capacity is 4TB and each meta segment is 16KB. As in §3, the average sizes of keys and values are 32B and 1KB, respectively [5]. Each entry in a meta segment is 36B (32B key and 4B pointers). A 16KB meta segment can hold 398 <key, pointer> pairs. In a 4TB SSD, there exist 2^{32} 1KB objects, and thus the number of meta segments in the flash is about 10.8M (= $2^{32}/398$). Each of these must be pointed to by some level lists. Each level list entry is 8 B, and each entry has a corresponding start key whose average size is 32B. Thus, only 432MB (= 10.8M×(8B + 32B)) DRAM is needed to hold all the level lists.

4.2 Improving I/O Speed with Level Pinning

Eliminating Read Tails. Retrieving a KV object from PinK requires multiple flash lookups. In the worst case, $O(h-1)$ flash lookups are required to access a KV object. Bloom filters are typically used to avoid useless lookups on levels that do not have desired keys [14, 15]. As pointed out earlier, however, it cannot avoid long tails and causes high CPU costs.

In order to guarantee worst-case latency and to get rid of Bloom filters, PinK adopts *level pinning*. The idea of the level pinning is straightforward. If the LSM-tree has h levels, PinK keeps meta segments for top-k levels ($k \leq h-1$) in DRAM. This simple technique greatly reduces read tails. To process `GET()`, it first searches for a key in top-k levels in DRAM. Only when a key is not found in memory, it looks up the rest of levels resident in the flash. With the level pinning, the number

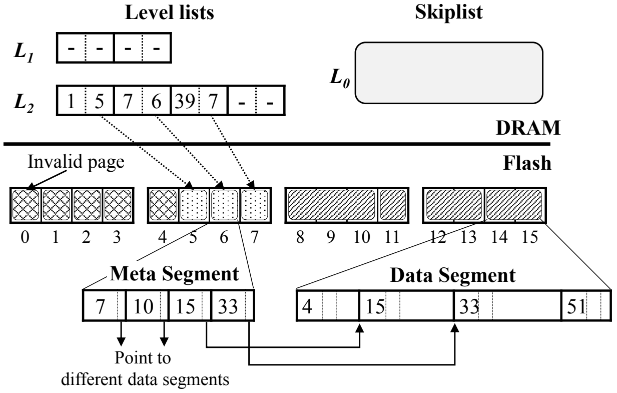


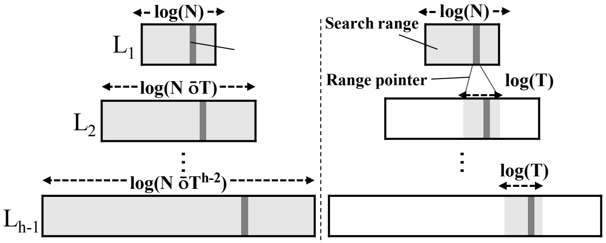
Figure 5: The DRAM and flash layouts of PinK after L_0 (in Figure 4) is flushed out with compaction. The tree hierarchy and KV objects are identical to those of Figure 2(b).

of the worst-case flash lookups is reduced to $O(h-k-1)$.

Level-pinning Memory Requirement. One might think that the level pinning would require large amounts of DRAM, but this is not the case. In the LSM-tree, a upper level (L_i) is T times smaller than a lower level (L_{i+1}), which implies that the level size increases *exponentially* by a factor of T. In the 4TB SSD organized with 5 levels, the amount of DRAM required to pin meta segments for L_1, L_2, L_3, and L_4 are 0.91MB, 50.86MB, 2.83GB, and 161.63GB, respectively. Meta segments for L_1, L_2, and even L_3 can be loaded in DRAM, considering a large controller DRAM of an SSD (*e.g.,* 4GB DRAM for 4TB SSD). The data structures of PinK do not require large amounts of DRAM (*e.g.,* 432MB), which enables us to pin more levels.

Reducing Compaction I/Os. Another benefit of the level pinning is that it eliminates flash I/Os involved in compaction. The level pinning maintains the meta segments of specific levels in DRAM. Thus, PinK does not need to issue any I/Os since pinned meta segments can be updated in DRAM directly. Dirty segments do not need to be written back to the flash because they are protected by capacitors.

To understand its benefit, let us consider how PinK performs compaction using the examples in Figures 4, and 5. Figure 5 is the data layout after the compaction. We assume that L_1 is pinned to DRAM. Before flushing out L_0, PinK fetches the corresponding meta segments from L_1 (*i.e.,* the page 0 in Figures 4 and 5) and sorts KV indices of L_0 and L_1, which creates two sorted meta segments. The sorted meta segments are then flushed out to L_1 (*i.e.,* the pages 3 and 4 in Figure 5). The level lists are updated accordingly. PinK recognizes that L_1 becomes full, and thus flushes out L_1 to L_2. To do this, PinK reads two meta segments from each of L_1 and L_2 (the pages 1∼4 in Figure 5), sorts them, and finally writes three sorted segments to L_2 (*i.e.,* the pages 5∼7). Since L_1 is pinned, PinK eliminates 3 reads and 2 writes out of 5 reads and 5 writes which occurs while conducting the compaction.



(a) level list search (b) Optimized level list search

Figure 6: Search path optimization with range pointers

4.3 Optimizing Search Path

When the LSM-tree retrieves a KV pair, it has to perform binary search on level lists until it finds a matching meta segment for a given key. This does not cause a serious overhead for higher levels (*e.g.,* L_1 and L_2) whose level lists have few entries. On the other hand, since level lists belonging to lower levels (*e.g.,* L_{h-1}) have many entries, the search overhead becomes huge. Figure 6 (a) shows how the LSM-tree performs binary search on the level lists. Suppose L_1 has N entries. Because a level size increases by a factor of T, L_2 has $N \cdot T$ entries, L_3 has $N \cdot T^2$ entries, and finally L_{h-1} has $N \cdot T^{h-2}$ entries. The worst-case time complexity of computation is thus expressed as $O(h^2 \cdot \log(T))$. The conventional LSM-tree with Bloom filters offers much lower computation time on average because it is able to skip binary search operations on unnecessary levels by performing membership tests first. PinK does not use Bloom filters and thus, cannot exploit this

To reduce the search overhead, PinK uses two techniques. The first one is to reduce string comparison costs by using a prefix of a key. Recall that each entry of a level list has two pointers, each of which points to a meta segment and a start key string, respectively. We further include a prefix which holds exactly the first four bytes of a start key. During binary search, PinK compares the first four bytes of an input key with a prefix. Only when they match, it performs a full string comparison using the pointer to the key.

The second one is to reduce search ranges of the level lists by borrowing fractional cascading technique [10]. Each entry of a level list now has another 4-byte pointer, called a range pointer. It locates the next lower level's entry which has the greatest start key but whose key is less than or equal to that of the upper level entry. Given a key to search, PinK does binary search for L_1 and finds an entry, say $e^i_{L_1}$, in L_1's level list. If a meta segment pointed to by $e^i_{L_1}$ does not have a matched KV index, PinK has to go down to the next level, L_2. The range pointer of $e^i_{L_1}$ becomes the lower search bound for L_2. Then, the range pointer of the next entry $e^{i+1}_{L_1}$ in the same level (*i.e.,* L_1) is the upper search bound. As shown in Figure 6 (b), using two pointers, the number of entries we have to do binary search in L_2 is reduced to T, on average, since a level size increases by a factor of T. This can be applied for lower levels, L_3, ... , L_{h-1}. Thus, the average time complexity reduces to $O(h \cdot \log(T))$.

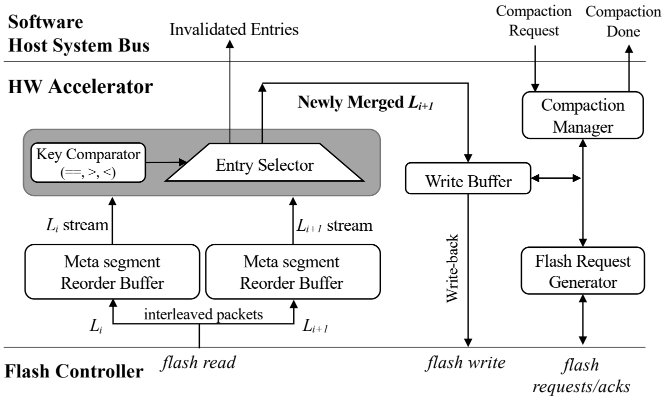


Figure 7: Compaction accelerator for flash-resident levels

With prefixes and range pointers, each entry size in the level lists increases to 16 bytes from 8 bytes. Fortunately, the lowest level L_{h-1}, which has the largest entries, does not maintain range pointers. As a result, additional DRAM is about 43.9MB in the same setting as in §4.1.

4.4 Speeding up Compaction

While the level pinning effectively reduces the number of I/Os for compaction, it does not remove the computation cost for sorting KV pairs. We address this problem by offloading some compaction tasks to a special HW accelerator in the SSD controller. The idea behind this is that compaction is just like merging two sorted lists of KV indices into a single sorted list. The HW accelerator placed between the flash and the host data bus can easily merge two flash-resident levels as meta segments of two levels are streamed from flash at wire speed. The accelerator writes the merged meta segments back to the flash without CPU involvement. By using the HW accelerator, we not only alleviate the computation overhead but improve I/O bus utilization since no to-be-merged and merged segments transferred over system bus. Remaining I/O bandwidth can be utilized by DRAM and flash for other tasks such as searching upper levels or managing pinned levels.

Figure 7 describes the architecture of the HW accelerator. We briefly present how the compaction accelerator for flash-resident levels works. The PinK software requests the accelerator to perform compaction by providing lists of the meta segments' flash addresses of two levels (L_i and L_{i+1}) to be merged and a list of flash addresses to which the merged meta segments (L_{i+1}) are written back. The flash request generator schedules multiple read requests to maximize the flash bandwidth utilization. Since the packets of different flash channels are interleaved, we need to use per-channel reorder buffers for each level to serialize the stream of meta segments.

Once we have sorted meta segment streams from two levels, the compaction engine (gray box in Fig. 7) only needs to keep comparing the keys of two levels and emitting the smaller one. The accelerator generates the output stream at wire speed without any computation overhead. When two keys match, the entry from the upper level (L_i) supersedes as it is more recent one. Note that the accelerator informs the software the metadata of invalidated entries from L_{i+1} for various purposes such as garbage collection. The generated merged meta segment stream (L_{i+1}) is written back to the flash via small write buffers. Once the operation completes, the accelerator responds with the number of flash pages consumed by the newly generated L_{i+1} meta segments so that the software can reclaim unused flash addresses previously provided to the accelerator.

While not shown in detail, we have a similar accelerator for merging pinned levels that reads from and writes back to host DRAM. DMA engines are used instead of a flash request generator and we do not need reorder buffers.

4.5 Optimizing Garbage Collection

The LSM-tree appends all the data to the flash. As compaction is repeated, obsolete data, which are no longer referenced to by the tree, are accumulated in the flash and must be erased by GC later. There are roughly two types of obsolete data that are created by compaction. The first type is old meta segments. While performing compaction, PinK writes new meta segments that replace old ones. For example, the meta segments stored in the pages 0, 1, and 2 in Figure 5 are not managed by the tree anymore since they contain old indices. The second type is an outdated KV object which was updated with a new one or removed by a client. Outdated KV indices are discarded from the LSM-tree during compaction (see §4.4) so that no meta segments point to them. But, their KV data are still stored somewhere in a data segment(s).

To erase obsolete data and to keep maintaining free space, PinK triggers GC when free space is nearly exhausted. It selects a victim flash block, copies valid data (*i.e.,* pages or KV pairs) to a free block, and erases the victim. For hot-cold separation, meta segments are isolated in different blocks from data segments. PinK should perform GC differently depending on the type of blocks selected as a victim.

GC for Meta Segment: If a victim block to GC is a meta-segment block and thus has only meta segments, PinK retrieves a start key of a meta segment by reading its page. Then, it looks up the level lists to see if there is any entry pointing to it. If not, PinK skips it since that segment is obsolete (*e.g.,* the page 4 in Figure 5). Otherwise (*e.g.,* the page 5), it moves the page (*i.e.,* meta segment) to a free page, and then updates the entry so that it locates a new flash page. Cleaning meta segments is cheap because it involves valid page copies and updates of the level lists in DRAM.

GC for Data Segment: Cleaning a data-segment block requires more efforts. Each data segment keeps metadata (*i.e.,* keys and sizes) as noted in §4.1. By scanning a data segment from the victim block, PinK extracts keys for values to move for GC. Using these numbers, PinK looks up the level lists and finds associated meta segments to check the validity (valid or not) of each value. If a meta segment is not pinned in DRAM, it must be read from the flash. In this way, PinK collects a list of valid values in the victim.

The simplest approach to reclaim free space, which is used by Wisckey, is to copy valid values to free pages and to erase the victim block. The meta segments associated with the values should be updated and flushed out to the flash so that they point to the new locations of the values. For meta segments pinned in DRAM, no flash writes are necessary. This approach, however, creates many updates on meta segments in the flash. We observe that many victim values are associated with flash-resident meta segments because they were written long time ago and their meta segments were likely to be demoted to lower levels. Moreover, only few values belong to the same meta segment (*e.g.,* 1$\sim$2 values, on average, in random write workloads). Thus, to move only 1$\sim$2 values, one meta-segment update is required.

To avoid this, PinK takes an approach that delays updates of meta segments in the flash. PinK writes valid KV pairs to L_0 again and then just erases the victim block. Corresponding meta segments now point to wrong flash pages erased by GC, but this is not a problem at all. Read requests to the rewritten KV pairs are served by higher levels, and old entries in the meta segments will eventually be discarded during compaction later. This approach slightly increases compaction costs, but greatly reduces GC costs by reducing meta segment updates. This is because victim KV pairs rewritten to L_0 are coalesced with neighboring KV pairs and then are written to the same meta segment together.

Note that since KV pairs sitting in lower levels are moved to L_0 during GC, it possibly hurts read latency. However, it does not affect the worst-case read latency, which is one of our design goals, because it is bounded by the number of pinned levels.

4.6 Durability and Scalability Issues

Durability with Limited Capacitor: We have assumed that a built-in capacitor in the device can protect the entire DRAM. However, unlike high-end enterprise SSDs, some SSDs do have not enough capacitors to protect all the metadata in DRAM. Running PinK on such devices results in a metadata durability issue. PinK can address this issue by regularly writing data structures that reside in DRAM (*e.g.,* level lists, pinned levels, and L_0) into flash. L_0 can be durable by logging incoming write requests into a log area in the flash before processing the requests. Hash-based KV-SSD also needs to do the same task if its write buffer is not backed by capacitors. Level lists and pinned levels become dirty after compaction is conducted. PinK should flush out newly created level lists as well as pinned levels to the flash to make them persistent. Hash-based KV-SSD has to write dirty KV indices to in-flash buckets as well whenever they are updated. The metadata flush operation of PinK would be cheaper than that of hash-based KV-SSD, thanks to the write-optimized structure of LSM-tree.

DRAM Scalability: We have also assumed that the size of DRAM scales as the size of flash in SSDs (*i.e.,* DRAM capacity is kept 0.1% of the total capacity of flash). As mentioned in Section 2.5, the size of DRAM scales slower than that of flash. Thus, SSDs might ship with insufficient DRAM to pin all the top levels. This problem can be resolved by pinning fewer levels. It increases the worst-case index lookup cost, but PinK offers better worst-case performance than the hash and the conventional LSM-tree. The details are discussed in the last experiment of Section 5.2 Another option we can consider is reducing the height of the tree so that all the levels, except for the last one, can fit into DRAM. This sacrifices write performance owing to increased compaction cost, but bounds $O(1)$ lookup cost in the worst case.

5 Experiments

We present experimental results on PinK. Particularly, we seek to answer the following questions: (*i*) Does the level pinning improves both read latency and write throughput along with shorter tails? (*ii*) Is the HW sorter effective to reduce the compaction cost? (*iii*) What is the impact of GC on performance?

5.1 Experimental Setup

We have implemented PinK on our FPGA-based SSD platform with quad-core ARM Cortex-A53 (Xilinx ZCU102 [48]). The FPGA is used to implement HW accelerators and flash chip controller. The SSD platform has a 256GB custom flash array card. The size of a page is 8 KB, and the number of pages per block is 256. (See §3.2 for more detailed performance numbers.) It is connected to a host through 10 GbE (1.25 GB/s) whose bandwidth is high enough to saturate the maximum throughput of the flash array card. The I/O queue depth is set to 64, which is sufficient to fully utilize the parallelism of 8-channel and 8-way in our flash array card. We scale down the SSD capacity to 64GB, and DRAM for KV indexing structures (*e.g.,* the level lists and pinned meta segments) is set to 64MB – 0.1% of the SSD capacity.

We evaluate PinK using seven workloads from YCSB, a realistic cloud benchmark [13]. The details of the workloads are described in Table 1. Default key and value sizes are set

Table 1: A summary of YCSB workloads

	Load	A	B	C	D	E	F
R:W ratio	0:100	50:50	95:5	100:0	95:5	95:5	50:50* (*RMW)
Query type	Point					Range	Point
Request distribution	Uniform	Zipfian			Latest [13]	Zipfian	

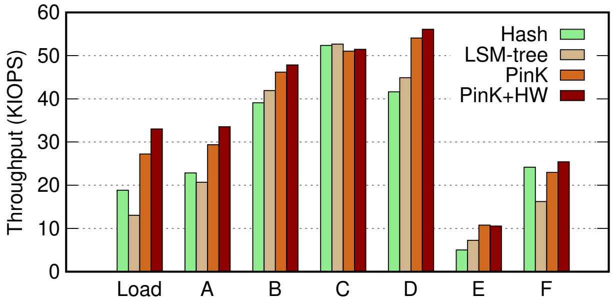
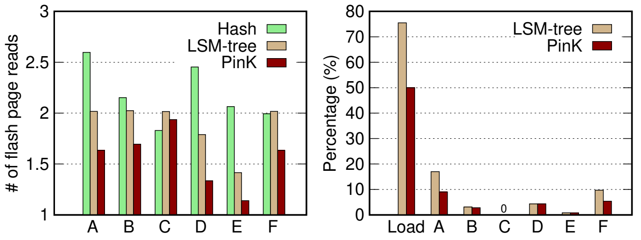


Figure 8: Overall throughputs of the four KV-SSD setups



(a) Flash page reads per query

(b) Compaction I/Os

Figure 9: The impact of the level pinning on flash read I/Os (a) and compaction I/Os (b)

to 32B and 1KB, respectively, which represent averages of common KV workloads [5]. For evaluation, we first created a 44GB KV pool on the 64GB SSD ('Load' in Table 1) – total 44M unique KV pairs are written. Then, we ran each workload ('A'∼'F' in Table 1) which sends 10M KV requests to the loaded data set. We initialized the SSD with the Load phase before any other workload executed. On the host, 64 YCSB clients ran simultaneously to maximize throughput. With 44GB data, the storage utilization was 69%. We assigned 10% of the SSD capacity (*i.e.,* 6.4GB), for over-provisioning.

To compare with PinK, we have implemented a hash-based KV-SSD based on what we described in §2.3. The KV-SSD denoted by `Hash` uses an 8-bit signature for each KV pair to balance a hash-table size and a signature collision rate. Note that, in our experimental setup with a relatively small data set, the 8-bit signature is large enough to provide a low collision rate. It requires 320MB of the hash table, which is much larger than the 64MB of DRAM for indexing. Therefore, `Hash` keeps only popular buckets in DRAM using the LRU replacement policy. `Hash` uses additional 1MB DRAM for a write buffer.

We compare `Hash` with two PinK configurations: one with no HW accelerator (`PinK`) and the other with HW accelerators (`PinK+HW`). The conventional LSM-tree implementation based on LightStore [11] (`LSM-tree`) is included for our evaluation. `LSM-tree` is equivalent to PinK, except that it does not employ the optimization techniques explained from §4.2 to §4.5. For `PinK`, `PinK+HW`, and `LSM-tree`, the number of total levels is set to 5. `PinK` and `PinK+HW` pin top-3 levels, $k = 3$. The meta segment size is the same as an 8KB page size. With 8KB meta segments, the amounts of DRAM for the level lists is 10MB (including both prefix and range pointers). The rest of DRAM, 54MB, thus can be used to pin levels. `LSM-tree` uses 9MB for level lists and 55MB of DRAM for bloom filters. As in `Hash`, for L_0 (a write buffer), 1MB DRAM is additionally assigned to `PinK`, `PinK+HW`, and `LSM-tree`.

5.2 Performance Analysis

YCSB Throughput: We measured IOPS of the four KV-SSD setups (`Hash`, `LSM-tree`, `PinK`, and `PinK+HW`) using YCSB. Figure 8 shows the results. `PinK+HW` outperformed `Hash` and `LSM-tree`, providing 37% and 44% higher throughputs, on average, respectively. `LSM-tree` suffered seriously from high CPU overheads caused by rebuilding bloom filers as well as sorting KV pairs. By eliminating bloom filters and reducing compaction I/Os, `PinK` improved IOPS by 34%, on average, over `LSM-tree`. Using the HW accelerators for sorting further improved the performance. As depicted in Figure 8, `PinK+HW` achieved 7.2% higher IOPS than `PinK` on average.

Those benefits of PinK were evident for the workloads with many writes. For Load, YCSB-A, and YCSB-F, we observed that `PinK+HW` improved IOPS by 56∼152% and 10∼21% over `LSM-tree` and `PinK`, respectively. Even with the workloads having relatively small writes (*i.e.,* YCSB-B, D), `PinK+HW` exhibited 14∼24% and 3% higher IOPS than `LSM-tree` and `PinK`, respectively. For the read-only workload, YCSB-C, no performance benefits were observed with `PinK` and `PinK+HW`.

One of the observations we did not expect was that PinK significantly outperformed `LSM-tree` for YCSB-D which issues only a small number of writes. This was due to the somewhat unique I/O behavior of YCSB-D that read recently-written KV pairs frequently. In PinK, recently-written KV pairs were stored in top levels pinned to DRAM. Thus, the majority of `GET()` requests were directly served by pinned levels, avoiding flash I/Os.

`LSM-tree` performed worse than `Hash` for the write-intensive benchmarks (Load, YCSB-A and F) owing to CPU overheads, but exhibited higher IOPS for the read-oriented workloads (YCSB-B, C and D). For YCSB-E with range queries, the LSM-tree-based KV-SSDs showed much higher IOPS than `Hash`, thanks to their sorted indexing structure.

Impact of Level Pinning: Figure 9 shows the impact of the level pinning on read and write I/O counts. As shown in Figure 9(a), `PinK` reduced the number of flash reads per query by 33% and 62% over `LSM-tree` and `Hash`, respectively. Since `PinK` pinned exact KV indices in DRAM, it eliminated many flash reads.

`Hash` was badly affected from hash misses and collisions. `Hash` maintained only signatures in DRAM. Thus, even when it has hits on `SET()` requests, it had to retrieve exact keys from flash unless designated buckets were empty. `LSM-tree` exhibited two flash page reads per query: one for a KV index and the other for a value (1 KB). This is because Monkey bloom filters [14] used in `LSM-tree` requires one read to fetch

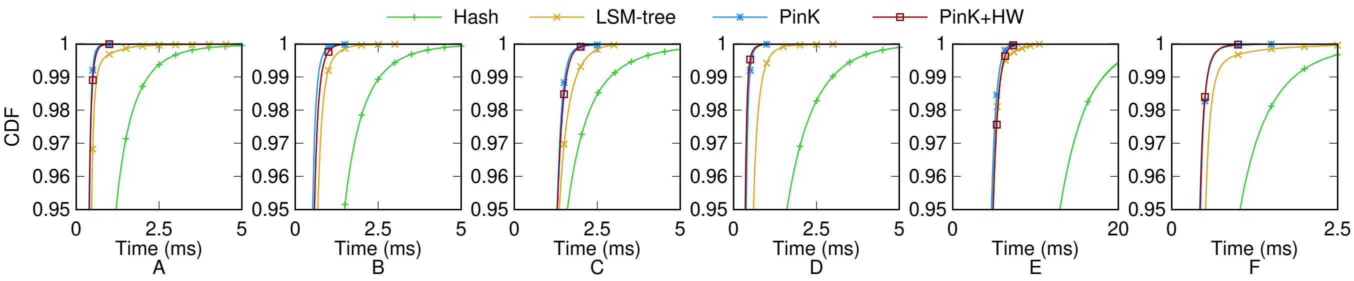


Figure 10: CDF graphs of read latency of `Hash`, `LSM-tree`, `PinK`, and `PinK+HW` under YCSB

indices, on average. For YCSB-D and E, the number of reads per query was less than 2. Since YCSB-D tends to read recently written KV pairs, many of `GET()`s were directly served by L_0 or pinned levels. YCSB-E contained range queries, so `LSM-tree` could fetch several desired KV indices by one read.

Figure 9(b) shows the percentage of compaction I/O out of the total I/O for LSM-tree operations (both reads and writes). By absorbing many index updates in pinned levels, it reduced the number of compaction I/Os by 52% over `LSM-tree`. Except for Load and YCSB-A with many writes, compaction I/Os only accounted for less than 20% of the total I/Os. However, as shown in Figure 8, the negative impact of compaction I/O ratio on the throughput was significant.

YCSB Read Latency: Figure 10 shows CDF graphs of read response times of the four KV-SSD setups. Table 2 also lists average, 99^{th}, 99.9^{th}, and 99.99^{th} percentile read latency of `Hash`, `LSM-tree`, and `PinK`. As expected, `PinK` and `PinK+HW` showed better average latency with shorter tails compared to the others. Thanks to bloom filters, `LSM-tree` performed fairly well compared to hash-based one, but had long tails as expected. `Hash` suffered from long tails due to multiple flash I/Os caused by hash misses and collisions. YCSB-E showed longer latency than the others because it issued range queries that carry multiple `GET()` commands.

Impact of Search Path Optimization: To understand the impact of the search path optimization, we carried out experiments with optimization techniques enabled one by one. `NO-OPT` represents PinK with no optimization, `Range` is PinK with range pointers, and `ALL` is with both range pointers and prefix. We used Load and YCSB-C workloads.

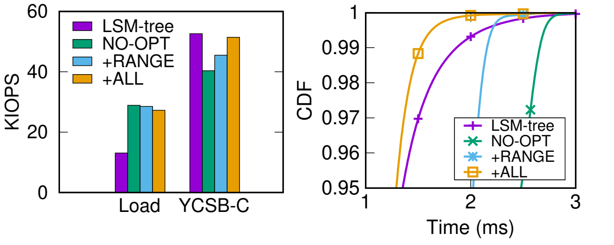
Table 2: Comparison of average and tail latency (unit: μs)

	Percentile	A	B	C	D	E	F
`Hash`	Average	410	573	592	501	5,628	370
	99^{th}	2,180	2,550	2,900	3,030	17,550	1,850
	99.9^{th}	4,180	4,600	5,710	5,090	25,360	3,260
	99.99^{th}	9,430	9,340	9,830	7,530	34,420	5,180
`LSM-tree`	Average	302	395	722	294	3,142	329
	99^{th}	640	960	1,870	890	5,790	680
	99.9^{th}	1,700	1,630	2,680	1,370	8,800	1,890
	99.99^{th}	5,250	3,140	3,450	3,210	10,740	3,750
`PinK`	Average	236	290	732	161	3,027	248
	99^{th}	490	700	1,820	490	5,550	540
	99.9^{th}	670	1,040	2,180	720	6,640	800
	99.99^{th}	1,300	1,800	2,370	1,060	7,590	1,540

Figure 11 (a) shows the throughputs under Load and YCSB-C. For Load, there were slight performance drops as the optimization technique was added. This was due to overheads required for managing additional data structures. These were not significant. For YCSB-C with 100% reads, high throughput improvements were observed. In particular, `ALL` exhibited almost the same read throughput as `LSM-tree`. This means that the search overheads were almost eliminated. While `LSM-tree` has the same worst case computation time of $O(h^2 \cdot log(T))$ as that of `NO-OPT`, `LSM-tree` has better throughput because its Bloom filter can much improve the average computation overhead by skipping searching of many levels. Figure 11 (b) presents the CDF of read latency under YCSB-C. We observed similar performance trends. `ALL` showed almost the same read latency as `LSM-tree` but with shorter tails.

Garbage Collection: With all the workloads of YCSB, GC did not involve many valid page copies. This was because almost all of the victim blocks were meta-segment blocks that held invalid KV indices. To simulate a situation where GC severely triggered, we designed another set of experiments. We first created a KV pool with 44M unique KV pairs, and then ran a synthetic workload that issued 100M `SET()`s with uniformly random keys to overwrite existing KV pairs. WAF reached 3.27 and became stable with little fluctuation after 90M `SET()`s were issued. This indirectly confirms that we issued sufficient I/Os to induce heavy GC I/O traffic.

Figure 12 analyzes the number of page writes issued during GC. `Hash` involved a smaller number of page writes for GC than `PinK`. After moving valid flash pages, both `Hash` and `PinK` have to update in-flash hash buckets or meta segments



(a) Throughput

(b) Read Latency

Figure 11: Impact of search path optimizations

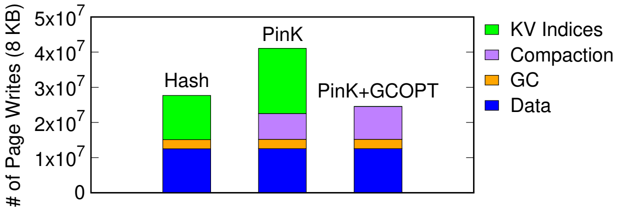


Figure 12: Analysis of GC cost: 'Data' represents pages written by `SET()`. 'GC' indicates pages written to move valid values for GC. 'Compaction' represents pages written to meta segments during compaction. 'KV Indices' indicates pages written to update meta segments or in-flash hash indices.

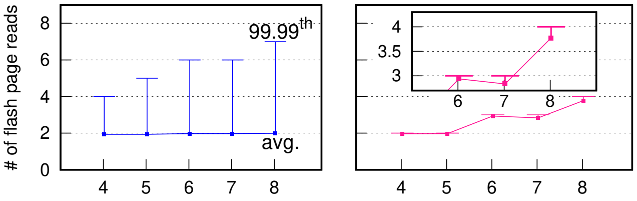
so that they point to the new locations of the moved pages (denoted by 'KV Indices' in Figure 12). Since a bucket size of `Hash` (8B signatures) is smaller than that of `PinK` (32B keys), more buckets are packed into a single flash page for `Hash`. Thus, the number of flash page I/O for updating KV indices becomes smaller than that of `PinK`. Even worse, `PinK` suffered from extra compaction I/Os.

`PinK+GCOPT` addresses this problem by rewriting victim KV pairs to L_0, instead of directly updating meta segments (see §4.5). This removed all flash writes associated with 'KV Indices', but potentially increased compaction costs since the indices for the victim pages in L_0 will be eventually written to meta segments again. This extra compaction cost was not so high. We observed that victim KV pairs in L_0 were likely to be coalesced with neighboring KV pairs and their indices were written to the same meta segment together.

Our results tell us that the compaction I/O cost of the LSM-tree, which is considered a major reason that makes people choose the hashing rather than the LSM-tree, is actually not a serious problem in achieving high I/O performance.

Read Latency and LSM-tree Height (h): Until now we have assumed that h and k are fixed to 5 and 3, respectively, and except for the last level, the rest is pinned to DRAM. As explained earlier (§4.2), this is a reasonable setup given that it required DRAM as small as 0.1% of flash storage and modern SSDs have more DRAM than that. However, to improve write performance further [31], one might want to increase the height of the tree. Unfortunately, as the tree gets taller, PinK cannot pin all the higher levels to DRAM. Given 64MB DRAM, for example, for h = 6, 7, and 8, the amount of DRAM required to pin all the levels but the last one are 176, 292, and 437MB, respectively. For h = 6 and 7, PinK cannot pin two lowest levels, and, for h = 8, the last three levels cannot be pinned. Even in such cases, the worst-case read latency can be bounded, but it increases to 3 reads (for h = 6 and 7) and 4 reads (for h = 8).

To understand its impact, using YCSB-C (100% reads), we measured the number of flash reads per query with various tree heights (h). Figure 13 shows the average read counts and 99.99^{th} percentile read counts of `LSM-tree` and `PinK`. The average read count of `LSM-tree` was close to 2. Again,



(a) Bloom filter (b) PinK

Figure 13: The number of flash page reads with varying h

regardless of h, Monkey required one flash read for fetching KV indices, on average. However, owing to its probabilistic nature, the tail latency increased greatly, and the gap between the tail and the average got wider as h increased.

Unlike `LSM-tree`, `PinK` exhibited stable read counts. While the average read count increased along with h, the worst-case read count was bounded as $O(h-k-1)$. For YCSB-C, there were no huge differences between the average and the tail read counts. This is because YCSB-C had low temporal locality and thus the majority of `GET()` were served by the flash-resident last level This experimental results confirm that PinK can provide more *stable* read latency even when h is set high and all the levels cannot be pinned to DRAM.

6 Conclusion

We have presented a novel LSM-tree-based KV-SSD design, called *PinK*. By pinning KV indices of top levels of the LSM-tree to DRAM, PinK is able to guarantee the worst-case read latency, while improving average read latency. Moreover, by combining the level pinning with hardware accelerators, PinK not only eliminated sorting overheads, but reduced I/O operations related to compaction greatly. Our experimental results show that PinK outperformed existing hash-based KV-SSDs in tail read-latency, average read-latency, and I/O throughput. In future, we plan to explore the idea of the level pinning in general-purpose KVS like RocksDB. We think the main challenge in realizing this idea is providing durability for pinned levels in the host system. Using emerging technologies such as persistent memory may offer a solution.

Acknowledgments

We would like to thank our shepherd, Dr. Rusty Sears, and four anonymous reviewers for all their helpful comments. We also thank Samsung Electronics for providing KV-SSD prototypes. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-11. Arvind and Chanwoo Chung was partially funded by NSF (CCF-1725303) and Samsung Semiconductor (GRO grants). PinK source code can be found at [1] and [2].

References

[1] PinK HW Source Code. https://github.com/chanwooc/lightstore-platform/tree/pink-hw.

[2] PinK SW Source Code. https://github.com/dgist-datalab/PinK.

[3] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S., and Panigrahy, R. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[4] Ashkiani, S., Li, S., Farach-Colton, M., Amenta, N., and Owens, J. D. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium* (2018), pp. 430–440.

[5] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[6] Axboe, J. FIO: Flexible I/O Tester Synthetic Benchmark. *URL https://github.com/axboe/fio (Accessed: 2015-06-13)* (2005).

[7] Bae, D.-H., Jo, I., Choi, Y. A., Hwang, J.-Y., Cho, S., Lee, D.-G., and Jeong, J. 2B-SSD: The Case for Dual, Byte- and Block-addressable Solid-state Drives. In *Proceedings of the Annual International Symposium on Computer Architecture* (2018), pp. 425–438.

[8] Bender, M. A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R. P., and Zadok, E. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment 5*, 11 (2012).

[9] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. Faster: A Concurrent Key-value Store with In-place Updates. In *Proceedings of the ACM International Conference on Management of Data* (2018), ACM, pp. 275–290.

[10] Chazelle, B., and Guibas, L. J. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica 1*, 1 (1986), pp. 133–162.

[11] Chung, C., Koo, J., Im, J., Arvind, and Lee, S. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 939–953.

[12] Colgrove, J., Davis, J. D., Hayes, J., Miller, E. L., Sandvig, C., Sears, R., Tamches, A., Vachharajani, N., and Wang, F. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the ACM International Conference on Management of Data* (2015), pp. 1683–1694.

[13] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing* (2010), pp. 143–154.

[14] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the ACM International Conference on Management of Data* (2017), pp. 79–94.

[15] Dayan, N., and Idreos, S. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM International Conference on Management of Data* (2018), pp. 505–520.

[16] Debnath, B., Sengupta, S., and Li, J. FlashStore: High Throughput Persistent Key-value Store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), pp. 1414–1425.

[17] Facebook, Inc. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org.

[18] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[19] Herlihy, M., Shavit, N., and Tzafrir, M. Hopscotch Hashing. In *International Symposium on Distributed Computing* (2008), Springer, pp. 350–364.

[20] IC Knowledge LLC. Lithovision-2020: Economics in the 3D Era. https://semiwiki.com/wp-content/uploads/2020/03/Lithovision-2020.pdf.

[21] Jin, Y., Tseng, H.-W., Papakonstantinou, Y., and Swanson, S. KAML: A Flexible, High-performance Key-value SSD. In *Proceedings of the*

IEEE International Symposium on High Performance Computer Architecture (2017), pp. 373–384.

[22] Jun, S.-W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S., and Arvind. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the Annual International Symposium on Computer Architecture* (2015), pp. 1–13.

[23] Kai Ren, G. G. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference* (2013).

[24] Kang, Y., Pitchumani, R., Mishra, P., Kee, Y.-s., Londono, F., Oh, S., Lee, J., and Lee, D. D. G. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the ACM International Conference on Systems and Storage* (2019), pp. 144–154.

[25] Kim, S.-H., Kim, J., Jeong, K., and Kim, J.-S. Transaction Support using Compound Commands in Key-Value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (July 2019).

[26] Kourtis, K., Ioannou, N., and Koltsidas, I. Reaping the Performance of Fast NVM Storage with uDepot. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2019), pp. 1–15.

[27] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), pp. 35–40.

[28] Lee, C.-G., Kang, H., Park, D., Park, S., Kim, Y., Noh, J., Chung, W., and Park, K. iLSM-SSD: An Intelligent LSM-tree based Key-Value SSD for Data Analytics. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2019), pp. 384–395.

[29] Lepers, B., Balmau, O., Gupta, K., and Zwaenepoel, W. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 447–461.

[30] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2016), pp. 133–148.

[31] Luo, C., and Carey, M. J. LSM-based Storage Techniques: a Survey. *The VLDB Journal* (2019).

[32] Mármol, L., Sundararaman, S., Talagala, N., Rangaswami, R., Devendrappa, S., Ramsundar, B., and Ganesan, S. NVMKV: A Scalable and Lightweight Flash Aware Key-value Store. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems* (2014).

[33] NGD Systems, Inc. NGD Catalina NVMe SSD. https://www.ngdsystems.com/products/, 2018.

[34] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), pp. 351–385.

[35] Pagh, R., and Rodler, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (2004), pp. 122–144.

[36] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), pp. 26–52.

[37] Samsung Electornics. Samsung Smart SSD. https://samsungatfirst.com/smartssd-ocp/, 2018.

[38] Samsung Electronics. KV SSD Host Software Package. https://github.com/OpenMPDK/KVSSD.

[39] Samsung Electronics. Samsung Introduces World's Largest Capacity (15.36TB) SSD for Enterprise Storage Systems. https://news.samsung.com/global/samsung-now-introducing-worlds-largest-capacity-15-36 2016.

[40] Samsung Electronics. Samsung Key Value SSD enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2017.

[41] Samsung Electronics. 860EVO SSD Specification. https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf, 2018.

[42] Samsung Electronics. KV SSD Firmware Introduction. https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf, 2018.

[43] Samsung Electronics. 960PRO SSD Specification. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/, 2019.

[44] SHEEHY, J., AND SMITH, D. Bitcask: A Log-structured Hash Table for Fast Key/value Data. *Basho White Paper* (2010).

[45] SNIA. Key Value Storage API Specification Version 1.0. https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[46] TWITTER INC. Fatcache: Memcache on SSD. https://github.com/twitter/fatcache.

[47] WANG, J., ZHANG, Y., GAO, Y., AND XING, C. pLSM: A Highly Efficient LSM-Tree Index Supporting Real-Time Big Data Analysis. In *Proceedings of IEEE Annual Computer Software and Applications Conference* (2013), pp. 240–245.

[48] XILINX. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2018.

[49] XU, S., LEE, S., JUN, S.-W., LIU, M., HICKS, J., ET AL. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment 10*, 4 (2016), pp. 301–312.

OPTIMUSCLOUD: Heterogeneous Configuration Optimization for Distributed Databases in the Cloud

Ashraf Mahgoub *Purdue University*
Alexander Medoff *Purdue University*
Rakesh Kumar *Microsoft*
Subrata Mitra *Adobe Research*
Ana Klimovic *Google Research*
Somali Chaterji *Purdue University*
Saurabh Bagchi *Purdue University*

Abstract

Achieving cost and performance efficiency for cloud-hosted databases requires exploring a large configuration space, including the parameters exposed by the database along with the variety of VM configurations available in the cloud. Even small deviations from an optimal configuration have significant consequences on performance and cost. Existing systems that automate cloud deployment configuration can select near-optimal instance types for homogeneous clusters of virtual machines and for stateless, recurrent data analytics workloads. We show that to find optimal performance-per-$ cloud deployments for NoSQL database applications, it is important to (1) consider heterogeneous cluster configurations, (2) jointly optimize database and VM configurations, and (3) dynamically adjust configuration as workload behavior changes. We present OPTIMUSCLOUD, an online reconfiguration system that can efficiently perform such joint and heterogeneous configuration for dynamic workloads. We evaluate our system with two clustered NoSQL systems: Cassandra and Redis, using three representative workloads and show that OPTIMUSCLOUD provides 40% higher throughput/$ and 4.5× lower 99-percentile latency on average compared to state-of-the-art prior systems, CherryPick, Selecta, and SOPHIA.

1 Introduction

Cloud deployments reduce initial infrastructure investment costs and provide many operational benefits. An important class of cloud deployments is NoSQL databases, which allow applications to scale beyond the limits of traditional databases [17]. Popular NoSQL databases such as Cassandra, Redis, and MongoDB, are widely used in web services, big data services, and social media platforms. Tuning cloud-based NoSQL databases for performance[1] under cost constraints is challenging due to several reasons.

First, the search space is very large due to VM configurations and database application configurations. For example, cloud services provide many VMs that vary in their CPU-family, number of cores, RAM size, storage, network bandwidths, etc., which affect the VM's $ cost. At the time of writing, AWS provides 133 instance types while Azure provides 146 and their prices vary by a factor of 5,000×. On the NoSQL side, there are many performance-impacting configuration parameters. For example, Cassandra has 25 such parameters and sub-optimal parameter setting for one parameter (e.g., the *Compaction method*) can degrade throughput by 3.4× from the optimal. On the cloud side too, selecting the right VM type and size is essential to achieve the best Perf/$.

Second, there is the need for *joint* optimization while taking into account the dependencies between the NoSQL-level and VM-level configurations. For example, our evaluation shows that the optimal cache size of Cassandra for a VM type `M4.large` (with 8GB of RAM) is 8× the optimal cache size for `C4.large` (with 3.75GB RAM). Additionally, larger-sized VMs do not always provide better Perf/$ [67] as they may overprovision resources and unnecessarily increase the $ cost.

Third, there are many use cases of cloud applications where the workload characteristics change over time, sometimes unpredictably, necessitating reconfigurations [7, 11]. A configuration that is optimal for one phase of the workload can become very poor for another phase of the workload. For example, in Cassandra, with a large working set size, reads demand instances with high memory, while writes demand high compute power and fast storage.

Changing the configuration at runtime for NoSQL databases, which are stateful applications (i.e., with persistent storage), has a performance impact due to the downtime caused to the servers being reconfigured. Therefore, for fast changing workloads, frequent reconfiguration of the overall cluster could severely degrade performance [41]. Consequently, deciding *which* subset of servers to reconfigure is vital to minimize reconfiguration performance hit and to achieve globally optimal Perf/$ while respecting the user's availability requirements. However, changing the configurations of only

[1] We use the standard metrics of throughput and (tail) latency for measuring database performance. Specifically, we target maximizing throughput normalized by price in $, i.e., performance-per-unit-$ or Perf/$ for short.

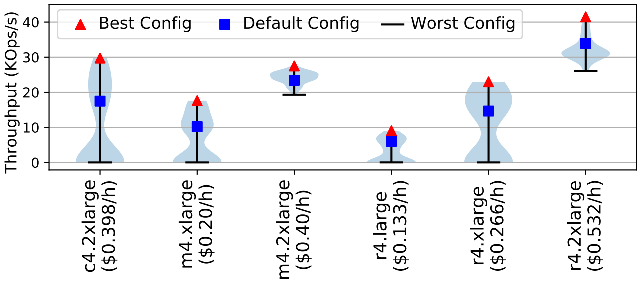
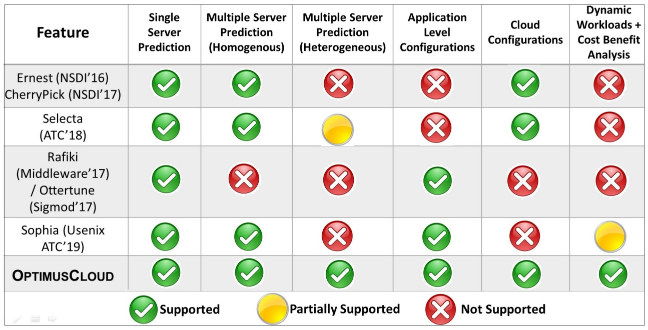


Figure 1: Violin plot showing performance (throughput) of Best, Default, and Worst database configurations across different EC2 VM types.



Feature	Single Server Prediction	Multiple Server Prediction (Homogenous)	Multiple Server Prediction (Heterogeneous)	Application Level Configurations	Cloud Configurations	Dynamic Workloads + Cost Benefit Analysis
Ernest (NSDI'16) CherryPick (NSDI'17)	✓	✓	✗	✗	✓	✗
Selecta (ATC'18)	✓	✓	◐	✗	✓	✗
Rafiki (Middleware'17) / Ottertune (Sigmod'17)	✓	✗	✗	✓	✗	✗
Sophia (Usenix ATC'19)	✓	✓	✗	✓	✗	◐
OptimusCloud	✓	✓	✓	✓	✓	✓
	✓ Supported		◐ Partially Supported		✗ Not Supported	

Table 1: OptimusCloud*'s key features vs. existing systems.*

a subset of servers naturally leads to *heterogeneous* clusters, which no prior work is equipped to deal with.

Existing Solutions: State-of-the-art cloud configuration tuners such as CherryPick [5] and Selecta [32] focus mainly on stateless, recurring workloads, such as big-data analytics jobs, while Paris [67] relies on a carefully chosen set of benchmarks that can be run offline to fingerprint which application is suitable for which VM type. Due to their target of static workloads and stateless jobs, a single cloud configuration is selected based on a representative workload and then fixed throughout the operation period. However, small workload changes can cause these "static tuners" to produce drastically degraded configurations. For example, a 25% increase in workload size with CherryPick makes the proposed configuration 2.6× slower than optimal (Section 5.4 in [5]). Also, in our experiments (Sec. 4.3), we find that CherryPick's proposed configuration for the write-heavy phase achieves only 12% of the optimal when the workload switches to a read-heavy phase. Hence, these *prior systems are not suitable for dynamic workloads*.

SOPHIA [41] addresses database configuration tuning for clustered NoSQL databases and can handle dynamic workloads. However, like the static tuner RAFIKI [40], SOPHIA's design focuses only on NoSQL configuration tuning and does not consider cloud VM configurations nor dependencies between VM and NoSQL configurations. Naïvely combining the NoSQL and VM configuration spaces causes a major increase in the search space size and limits SOPHIA's ability to provide efficient configurations (Sec. 3.5). Further, due to its atomic reconfiguration strategy (i.e., either reconfigure all servers or none), it suffers from all the drawbacks of the homogeneity constraint. Table 1 compares key features of OptimusCloud to various prior works in this field.

Our Solution: We introduce our system OptimusCloud, which jointly tunes the database and cloud (VM) configurations for dynamic workloads. **There are three key animating insights behind the design of OptimusCloud.** The *first* is that jointly tuning the database and cloud (VM) configurations for dynamic workloads is essential. To show how important this is, we benchmark one Cassandra server with a 30-min trace from one of our three workloads (MG-RAST) on 9 different EC2 VM types[2]. For each type, we use 300 different database configurations selected through grid search. We show the performance in terms of Ops/s for the best, default, and worst configurations in Fig. 1. We see a big variance in performance w.r.t. the database configurations—up to 74% better performance over default configurations (45% on average). Further, the best configurations vary with the VM type and size (for the 6 VM types shown here, there are 5 distinct best DB configurations). This emphasizes the need for tuning both types of configurations *jointly* to achieve the best Perf/$. The *second key insight* is that in order to optimize the Perf/$ for a dynamic workload, it is necessary to perform non-atomic reconfigurations, i.e., for only part of the cluster. Reconfiguration in a distributed datastore is a sequential operation (in which one or a few servers at a time are shutdown and then restarted) to preserve data availability [31, 41]. This operation causes transient performance degradation or lower fault tolerance. Reconfiguration is frequent enough for many workloads that this performance degradation should be avoided, e.g., MG-RAST has a median of 430 significant switches per day in workload characteristics. Accordingly, *heterogeneous configurations* have the advantage of minimizing the performance hit during reconfiguration. Further, in the face of dynamic workloads, there may only be time to reconfigure part of the overall cluster. Also, from a cost-benefit standpoint, maximizing performance does not need *all* instances to be reconfigured (such as to a more resource-rich instance type), rather a carefully selected subset. We give a simple example to make this notion concrete in Section 2. The *third key insight* is that for a particular NoSQL database (with its specifics of data placement and load balancing), it is possible to create a model to map the configuration parameters to the performance of each server. From that, it is possible to determine the overall heterogeneous cluster's performance. OptimusCloud leverages performance modeling to search for the optimal cluster configuration.

The workflow of OptimusCloud comprises offline training and online prediction and reconfiguration phases as shown in Fig. 2. At runtime, OptimusCloud takes user require-

[2] MG-RAST is the largest metagenomics portal and data repository and gets queries from across the globe which cause unpredictable read-write patterns to the backend Cassandra.

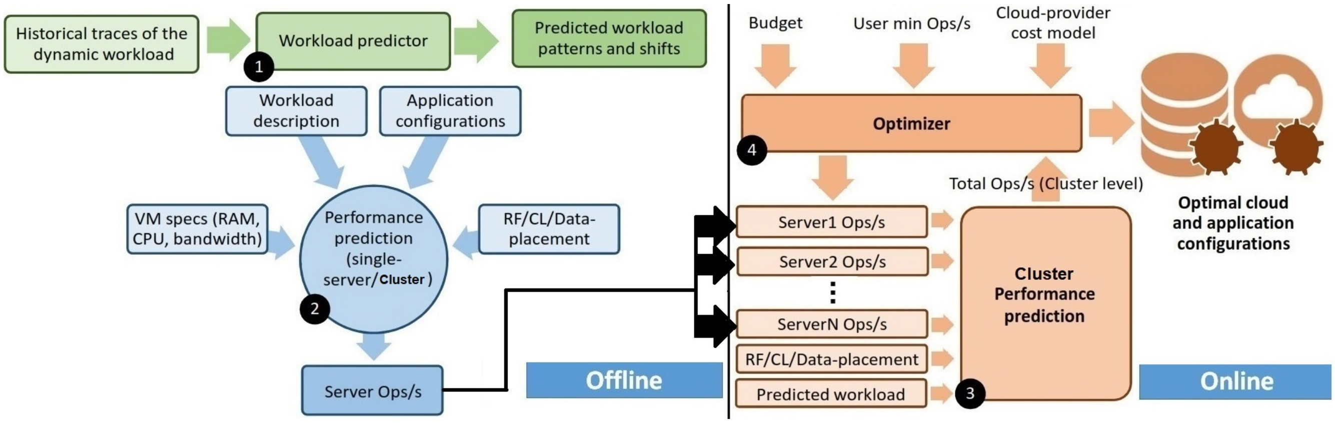

Historical traces of the dynamic workload
Workload predictor
1
Predicted workload patterns and shifts
Workload description
Application configurations
VM specs (RAM, CPU, bandwidth)
Performance prediction (single-server/Cluster)
2
RF/CL/Data-placement
Server Ops/s
Offline
Budget
User min Ops/s
Cloud-provider cost model
Optimizer
4
Total Ops/s (Cluster level)
Optimal cloud and application configurations
Server1 Ops/s
Server2 Ops/s
⋮
ServerN Ops/s
RF/CL/Data-placement
Predicted workload
3
Cluster Performance prediction
Online

Figure 2: Overview of OptimusCloud*'s workflow. First, a workload predictor is trained with historical traces from the database to be tuned. Second, a single server performance predictor is trained to map workload description, VM specs, and NoSQL application configuration to throughput. Third, a cluster-level performance predictor is used to estimate the throughput of the heterogeneous cluster of servers. In the online phase, our optimizer uses this predictor to evaluate the fitness of different VM/application configurations and provides the best performance within a given budget.*

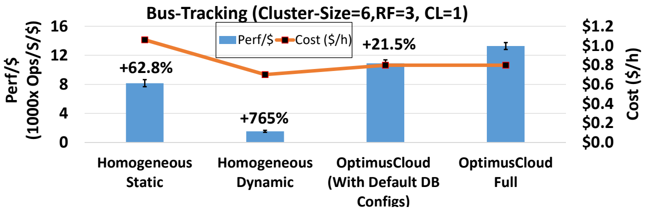


*Figure 3: Importance of creating heterogeneous clusters (*OptimusCloud *Full) over homogeneous clusters (both Static and Dynamic) for Bus-Tracking application. Tuning both application and cloud configurations (*OptimusCloud *Full) has benefit over tuning only the VM configuration (3rd bar from left). The percentage value on the top of each bar denotes how much* OptimusCloud *improves over that particular scheme.*

ments of budget, availability, and consistency. It then combines the performance model with a workload predictor and a cost-benefit analyzer to decide: when the workload changes sufficiently, what should be the new (possibly heterogeneous) configuration. It decides what minimal set of servers should be reconfigured.

Evaluation: We apply OptimusCloud to two popular NoSQL databases—Cassandra and Redis—and evaluate the system on traces from two real-world systems, and one simulated trace from an HPC analytics job queue. All three use cases represent dynamic workloads with different query blends. We evaluate the Perf/$ achieved by OptimusCloud and compare this to three leading prior works, CherryPick [5], Selecta [32], and SOPHIA [41]. Additionally, we compare ourselves to the best static configuration determined with oracle-like prediction of future workloads and the theoretical best. OptimusCloud achieves between 80-90% of the theoretical best performance for the 3 workloads and achieves improvements between 9%-86.5%, 18%-173%, 17%-174%, and 12%-514% in Perf/$ over Homogeneous-Static, CherryPick, Selecta, and SOPHIA respectively without degrading P99 latency (Sec. 4). Fig. 3 shows the improvement in Perf/$ due to OptimusCloud's heterogeneous configurations.

We make the following novel contributions in this paper.

1. We design a performance modeling-based technique for efficient *joint optimization* of database *and* cloud configurations to maximize the Perf/$ of a clustered database.

2. We design a technique to identify the minimal set of servers in a clustered database to reconfigure (concurrently) to obtain a throughput benefit. This naturally leads to heterogeneous configurations. To reduce the much larger search space that this causes, we design for a simplification that groups multiple servers that should be configured to the same parameters.

3. We show that OptimusCloud generalizes to two distinct NoSQL databases and different workloads, cluster sizes, data volumes, and user-specified requirements for replication and data consistency.

The rest of the paper is organized as follows. Section 2 gives the necessary background for the problem and a quantitative rationale. Section 3 describes the details of OptimusCloud's design. We evaluate OptimusCloud in Section 4 and survey related work in Section 5.

2 Background and Rationale

To evaluate the generalizability of OptimusCloud, we select two popular NoSQL databases with very different architectures.

2.1 Cassandra

Cassandra is designed for high scalability, availability, and fault-tolerance. To achieve these, Cassandra uses a peer-to-peer (P2P) replication strategy, allowing multiple replicas to handle the same request. Other popular datastores such as DynamoDB [20] and Riak [34] implement the same P2P strategy and we select Cassandra as a representative system from that category. Cassandra's replication strategy determines where replicas are placed. The number of replicas is defined as "Replication Factor" (RF). By default, Cassandra assigns an equal number of tokens to each node in the cluster where a token represents a sequence of hash values for the primary keys that Cassandra stores. Based on this token assignment, a Cassandra cluster can be represented as a ring topology [16]. Fig. 5 shows an example of 4 Cassandra servers and RF of 2.

2.2 Redis

Redis is an in-memory database and serves all requests from the RAM, while it writes data to permanent storage for fault tolerance. This design principle makes Redis an excellent choice to be used as a cache on top of slower file systems or datastores [54]. Redis can operate as either a stand-alone node or in a cluster of nodes [53] where data is automatically sharded across multiple Redis nodes. Our evaluation applies to the clustered mode of Redis. When a Redis server reaches the maximum size of its allowed memory (specified by the `maxmemory` configuration parameter), it uses one of several policies to decide how to handle new write requests. The default policy will respond with error. Other policies will replace existing records with the newly inserted record (the `maxmemory-policy` configuration parameter specifies which records will be evicted). The value of `maxmemory` needs to be smaller than the RAM size of the VM instance and the headroom that is needed is workload dependent (lots of writes will need lots of temporary buffers and therefore larger headroom). Thus, it is challenging to tune `maxmemory-policy` and `maxmemory` parameters with changing workloads and these two form the target of our configuration decision.

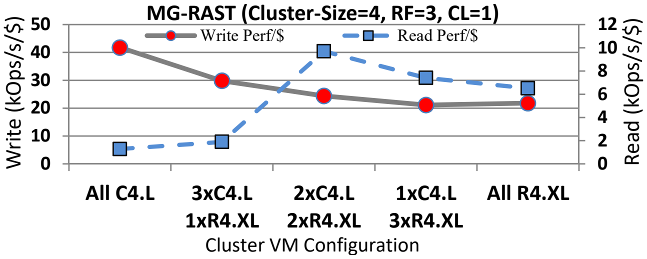


Figure 4: Change in Perf/$ for the write (solid) and read throughput (dotted) as we reconfigure the nodes from C4.large to R4.xlarge.

2.3 Example Rationale for Heterogeneous Configurations

Here we give a motivating example for selecting subset of servers to reconfigure. Consider a Cassandra cluster of 4 nodes with a consistency-level (CL[3]) = 1 and replication-factor (RF[4]) = 3, i.e., any pair of nodes has a complete copy of all the data. Also, assume that we only have two cloud configurations: C4.large, which is compute-optimized, and R4.xlarge, which is memory-optimized. C4.large is cheaper than R4.xlarge by 58% [8], whereas R4.xlarge has larger RAM (30.5GB vs 3.75GB) and serves read-heavy workloads with higher throughput. Now we test the performance of all possible combinations of VM configurations (All C4.L, 1 C4.L + 3R4.XL, ... etc.) for both read-heavy and write-heavy phases of the MG-RAST workload and show the saturation level throughput for each configuration in Fig. 4. The "All C4.large" configuration achieves the best write Perf/$ (41.7 KOps/s/$), however, it has the worst read Perf/$ (only 1.28 KOps/s/$) because reads of even common records spill out of memory. Now if two servers are reconfigured to R4.xlarge, the write Perf/$ decreases (24.4 KOps/s/$), while the read performance increases significantly (9.7 KOps/s/$), showing an improvement of 7.5× for read throughput over the all C4.large configuration. The reason for this huge improvement is Cassandra's design by which it redirects new requests to the fastest replica [19], directing all read requests to the two R4.xlarge servers. *Now we notice that switching more C4.large servers to R4.xlarge does not show any improvement in either reads or writes Perf/$, as the two R4x.large servers are capable of serving the applied workload with no queued requests.* This means that switching more servers will only reduce the Perf/$. Thus, the best Perf/$ is achieved by configuring to all C4.large in write-heavy phases, while configuring only 2 servers to R4x.large in read-heavy phases. **Therefore, heterogeneous configurations can achieve better Perf/$ compared to homogeneous ones under mixed workloads.**

3 Design

3.1 Workload Representation and Prediction

OptimusCloud uses a query-based model [4] to represent time-varying workloads. This model characterizes the applied workload in terms of the proportion of the different query types and the total volume of queries, denoted by W.

We use a workload predictor to learn time-varying patterns from the workload's historical traces, and predict the workload characteristics for a particular lookahead period. We notate the time varying workload at a given point in time t as $W(t)$. The task of the workload predictor is to provide OptimusCloud with $W(t+1)$ given $W(t)$,$W(t-1)$,... , $W(t-h)$, where h is the length of history. OptimusCloud then iteratively predicts the workload till a lookahead time l, i.e., $W(t+i), \forall i \in (1,l)$. We execute OptimusCloud with a simple Markov-Chain prediction model for both MG-RAST and Bus-tracking workloads while we have a deterministic fully accurate predictor for HPC. We do not claim any novelty in workload prediction and OptimusCloud is modular enough to easily integrate more complex estimators, such as

[3] CL: the minimum number of Cassandra nodes that must acknowledge a read or write operation before the operation can be considered successful

[4] RF: the total number of replicas for a key across a Cassandra cluster

neural networks [37, 39].

3.2 Performance Prediction

Combining NoSQL and cloud configurations produces a massive search space, which is impractical to optimize through exhaustive search. However, it is well known that not all the application parameters impact performance equally [40, 41, 62] and therefore OPTIMUSCLOUD reduces the search time by automatically selecting the most impactful parameters. Further, there exist dependencies among parameters, such as the dependency between the VM type (EC2) and Cassandra's *file-cache-size* (FCS) (Fig. 7). OPTIMUSCLOUD uses *D*-optimal design [48] to optimize the offline data collection process for training our performance model. *D*-optimal design answers this question: "*Given a budget of N data points to sample for a workload, which N points are sufficient to reveal the dependencies between configuration parameters?*". We experimentally determine that the significant dependencies in our target applications are at most pairwise and therefore we restrict the search to linear and quadratic parameters. We create a set of filters for feasible combinations of parameter values by mapping each parameter to the corresponding resource (e.g., *file-cache-size* parameter is mapped to RAM). Afterward, we check that the sum of all parameters mapped to the same resource is within that resource limit of the VM (e.g., the total size of all Cassandra buffer memories should not exceed the VM instance memory). We feed to *D*-optimal design the budget in terms of the number of data points that we can collect for offline training.

After collecting the data points determined by the *D*-optimal design, we train a random forest to act as a regressor and predict the performance of a single NoSQL server for any given set of configuration parameters, both database and VM. The average output of the different decision trees is taken as the final output. We choose random forest over other prediction models because of its easily interpretable results [50] and it has only two hyper-parameters to tune (`max_depth` and `forest_size`) compared to black-box models such as DNNs. OPTIMUSCLOUD trains a second random forest model to predict the overall cluster performance, using the predicted performance for each server, RF, CL and data-placement information. For both random forests, we use 20 trees and a maximum depth of each as 5 as that gives the best result within reasonable times.

3.3 Selection of Servers to Reconfigure

Selecting the right servers to reconfigure in a cluster is essential to achieve the best Perf/$. We introduce the notion of *Complete-Sets* to determine the right subset of servers to reconfigure. **We define a *Complete-Set* as the minimum subset of nodes for which the union of their data records covers all the records in the database at least once.**

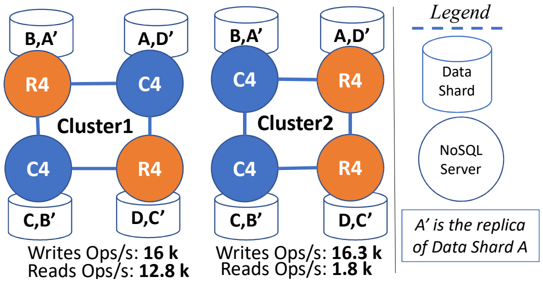


Figure 5: (RF=2, CL=1) Cluster performance depends not just on the configuration of each server, but also on the relative positions of the instances on the token ring. Cluster1 achieves 7× reads Ops/s over Cluster2 with the same VM types and sizes.

To see why the notion of *Complete-Set* is important, consider the two clusters shown in Fig. 5. Both clusters 1 and 2 use 2 C4.large and 2 R4.large and hence have the same $ cost. However, *Cluster1* achieves 7× the read Ops/s compared to *Cluster2*. The reason for the better performance of *Cluster1* is that it has one *Complete-Set* worth of servers configured to the memory-optimized R4.large architecture and therefore serves all read requests efficiently. On the other hand, *Cluster2*'s read performance suffers since all read requests to shard `B` (or its replica `B'`) have to be served by one of the C4.large servers, which has a smaller RAM and therefore serves most of the reads from desk. Accordingly, read requests to shards `B` or `B'` represent a bottleneck in *Cluster2* and cause a long queuing time for the reading threads, which brings down the performance the *entire* cluster for all the shards.

This means that all the servers within a Complete-Set must be upgraded to the faster configuration for the cluster performance to improve. Otherwise, the performance of the Complete-Set will be bounded by the slowest server in the set. OPTIMUSCLOUD partitions the cluster into one or more *Complete-Sets* using the cluster's data placement information. To identify the Complete-Sets, we collect the data placement information for each server of the cluster. OPTIMUSCLOUD queries this information either from any server (such as in Cassandra, using `nodetool ring` command) or from one of the master servers (such as in Redis, using `redis-cli cluster info` command). In Redis, identifying the Complete-Sets is easier since data tokens are divided between the master nodes only, while slaves have exact copies of their master's data. Therefore, a *Complete-Set* is formed by simply selecting a single slave node for every master node.

Maintaining Data Availability: To maintain data availability during reconfiguration of a Cassandra cluster, at least *CL* replicas of each data record must be up at any point in time. This puts an upper limit on the number of *Complete-Sets* that can be reconfigured concurrently as $Count(Complete\text{-}Sets) - CL$.

We show that the number of Complete-Sets in a cluster is *not* dependent on the number of nodes in the cluster, but is a constant factor. This is because when the cluster size increases, the range of keys assigned to every node decreases

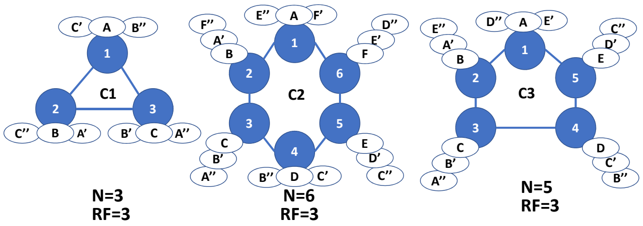


Figure 6: Replication examples with 3 different cluster sizes with RF=3. Cluster C1 has 3 nodes and each node has a complete copy of the data, therefore each node is a Complete-Set. Cluster C2 has 6 nodes and has the following Complete-Sets: [1,4] , [2,5] & [3,6]. Cluster C3 has 5 nodes (not divisible by RF=3), therefore it has two Complete-Sets: [1,3 (or 4)], [2,4 (or 5)].

and therefore the number of nodes that form a Complete-Set increases. This means that since OPTIMUSCLOUD reconfigures the instances in groups of one or more Complete-Sets concurrently, the total time to reconfigure a cluster is a constant factor independent of the cluster size. Figure 6 shows examples of *Complete-Set* for different cluster sizes

Property: OPTIMUSCLOUD **partitions the cluster into** S **Complete-Sets, and** S **is independent of the cluster size** N.

Proof. For a cluster of N servers with replication factor RF, there exists a total of RF copies of each record in the cluster, with no two copies of the same record stored in the same server. Assuming each node in the cluster is assigned an equal portion of the data (which NoSQL load-balancers try to achieve [35]), the size of a *Complete-Set* is $Size_{CompSet} = \lceil \frac{N}{RF} \rceil$. Consequently, the number of *Complete-Sets* in the cluster $S = \lfloor \frac{N}{Size_{compSet}} \rfloor$. If RF divides N, then the number of *Complete-Sets* is $S = \frac{N}{Size_{CompSet}} = RF$. Else, say $N\%RF = r$, then $S = \frac{RF}{1-r/N+RF/N}$, which is $\approx RF$ since in practice RF is not large, 3 being a practical upper bound. Thus, the number of *Complete-Sets* is independent of the cluster size and hence the reconfiguration time is also a constant.

□

Search Space Size Reduction: Heterogeneous configurations make the search space size much larger than with homogeneous configurations. Consider a cluster of N nodes and I VM options to pick from. If we are to pick a homogeneous cloud configuration for the cluster, we have I options. However, if we are to pick a heterogeneous cloud configuration, our search space becomes I^N. If we assume balanced data placement among the servers in the cluster (as clustered NoSQL databases are designed for), the search space becomes $C(N+I-1, I-1)$ (distribute N identical balls among I boxes). However, this search space size is still too large to perform an exhaustive search to pick the optimal configurations. A cluster of size N=20 nodes and I=15 VM options gives 1.3×10^9 different configurations to select from. One may use domain-specific insights about the domain to reduce the search space for specific applications [36] or for customized distributed strategies [28]. However we aim for generalizability here.

We use one insight about Complete-Sets to reduce the search space. The nodes within each Complete-Set should be homogeneous in their configuration. Otherwise, the performance of the Complete-Set will be equal to that of the slowest node in the set. *This means that the smallest atomic unit of reconfiguration is one Complete-Set.* This insight reduces the search space, while still allowing different *Complete-Sets* to have different configurations. Thus, the search space reduces to $C(S+I-1, I-1)$=680 configurations when $S = RF = 3$. **Also note that the configuration search space is constant rather than growing with the size of the cluster.**

3.4 Selecting the Reconfiguration Plan

3.4.1 Objective Function Optimization

The objective of OPTIMUSCLOUD is to find a reconfiguration plan that maximizes Perf/$ of the cluster under a given budget and with a minimum acceptable throughput. A *reconfiguration plan C* is represented as a time series of a vector of configurations (both NoSQL and VM):

$$\mathbf{C} = \left[\{C_1, C_2, \cdots, C_M\}, \{t_1, t_2, ..., t_M\}\right] \tag{1}$$

Where M is the number of steps in the plan and timestamp t_i represents how long the configuration C_i is applied. The lookahead is $t_L = \sum_{i=1}^{M} t_i$. The optimization problem is defined as:

$$\mathbf{C}^* =_{\mathbf{C}} \frac{f\left(\mathbf{W},\mathbf{C}\right)}{Cost\left(\mathbf{C}\right)} \tag{2}$$

$$\text{subject to } f\left(\mathbf{W},\mathbf{C}\right) \geq \text{minOps} \ \& \ Cost\left(\mathbf{C}\right) \leq \text{Budget}$$

Here, $f\left(\mathbf{W},\mathbf{C}\right)$ is the function that maps the workload vector $\mathbf{W}$ and the configuration vector $\mathbf{C}$ to the throughput (the cluster prediction model) and $\mathbf{C}^*$ is the best reconfiguration plan selected by OPTIMUSCLOUD. The two constraints in the problem prevent us from selecting configurations that exceed the budget or those that deliver unacceptably low performance.

The optimization problem described in Equation 2 falls under the category of gradient-free optimization problems [38], in which no gradient information is available nor can any assumption be made regarding the form of the optimized function. For this category of optimization problems, several meta-heuristic search methods have been proposed, such as, Genetic Algorithms (GA) , Tabu Search [64], and Simulated Annealing. We use GA due to two relevant advantages. First, constraints can be easily included in its objective function (i.e., the fitness function). Second, it provides a good balance between exploration and exploitation through crossover and mutation [59]. We use Python `Solid` library for GA [58] and `Scikit-learn` for random forests [55].

3.4.2 Cost-Benefit Analysis

Changing either NoSQL or cloud configurations at runtime has a performance cost due to downtime caused to nodes being reconfigured. We find that most of the performance-impacting NoSQL parameters (83% for Cassandra) necessitate a server restart and naturally, changing the VM type needs a restart as well. When a workload change is predicted in the online phase, OptimusCloud uses its performance predictor to propose new configurations for the new workload. Afterward, OptimusCloud estimates the reduction in performance given the expected downtime duration and compares that to the expected benefit of the new configurations. OptimusCloud selects configurations that maximize the difference between the benefit and the cost (both in terms of Throughput/$) . This cost-benefit analysis prevents OptimusCloud from taking greedy decisions, whenever the workload changes. Rather, it uses a long-horizon prediction of the workload over a time window to decide which reconfiguration actions to instantiate and when.

The benefit of the i^{th} step in the plan is given by:

$$B_{(i+1,i)} = \sum_{t \in t_{i+1}} f\left(W_t, C_{i+1}\right) - f\left(W_t, C_i\right) \tag{3}$$

where $f\left(W_t, C_{i+1}\right)$ is the predicted throughput using the new configuration C_{i+1}. The configuration cost is given by:

$$L_{(i+1,i)} = \sum_{p \in (C_i - C_{i+1})} t_{down} \times \delta_p \times f\left(W_t, C_i\right) \tag{4}$$

where p is any *Complete-Set* that is being reconfigured to move from configuration C_i to C_{i+1}, t_{down} is the expected downtime during this reconfiguration step, and δ_p is the portion of the cluster throughput that p contributes as estimated by our cluster predictor. We then normalize the benefit (Equation. 3) and the cost (Equation. 4) by the difference in price between configurations C_i and C_{i+1}. The value of t_{down} is measured empirically and its average value is 30 sec for NoSQL configurations and 90 sec for VM configurations.

3.5 Distinctions from Closest Prior Work

We describe the substantive conceptual differences of OptimusCloud from two recent, related works: Selecta and SOPHIA. OptimusCloud provides joint configuration tuning of both NoSQL and cloud VMs, while it considers heterogeneous clusters to achieve the best Perf/$. In Selecta, only heterogeneous cloud storage configurations are permissible (i.e., HDD, SSD, or NVMe). Accordingly, the configuration space in Selecta is much smaller and simpler to optimize using matrix factorization techniques. A simple extension of Selecta to our large search space produces very poor performance due to the sparsity of the generated matrix and the dependency between NoSQL and cloud configurations as we empirically show in Sec. 4.6.

In SOPHIA, only NoSQL parameters are configured and no computing platform parameters such as VM configurations are optimized. Even within NoSQL configurations, it only considers homogeneous configurations. Accordingly, SOPHIA makes a much simpler decision to either configure the complete cluster to the new configuration, or keep the old configuration—correspondingly its cost-benefit analysis is also coarse-grained, at the level of the entire cluster. For fast-changing workloads, it therefore often has to stick to the current configuration since there is not enough time to reconfigure the entire cluster (which needs to be done in a partly sequential manner to preserve data availability). *Similar to Selecta, a simple extension of SOPHIA to VM options cannot achieve the best Perf/$ for dynamic workloads, as it can only create homogeneous configurations across all phases of the workload.* We empirically show this in Sections 4.3 and 4.7.

4 Experimental Setup and Results

In this section, we evaluate OptimusCloud under different experimental conditions for the 3 applications. We deploy OptimusCloud and the datastore clusters (Cassandra or Redis) in Amazon EC2 in the US West (Northern California) Region. We also deploy a separate set of nodes in the same region to serve as workload generators (i.e., shooters). We vary the number of shooting threads in runtime to simulate the changes in the request rate in the workload trace. The results are averages of 20 runs, with each run using a different subset of the training data. Our evaluation answers four broad questions. (1) How does OptimusCloud compare in terms of Perf/$ and P99 latency with three state-of-the-art systems (which can only create homogeneous configurations) and two oracle-based baselines? (2) What is the accuracy of each module of OptimusCloud, such as, the workload and the performance predictors? (3) How do application requirements such as RF and CL impact OptimusCloud? (4) How does OptimusCloud generalize to different applications (we use three), databases (Cassandra and Redis), and levels of prediction errors?

Major Insights: We draw several key insights from our evaluation. **First**, the flexibility afforded by being able to reconfigure different parts of the cluster to different configurations is useful—all three prior protocols being compared (and in fact, all works to date) can only create homogeneous configurations. *Further, the proactive approach of initiating reconfiguration upon predicted workload change helps to handle dynamic workloads and keeps latency low (CherryPick and Selecta are both reactive).* **Second**, OptimusCloud's design reduces the heterogeneous configurations search space significantly. Accordingly, it is able to search efficiently and finds higher performing VM and NoSQL configurations than cluster configurations selected by CherryPick, Selecta, or SOPHIA. This improvement persists across applications (highest for the more predictable HPC analytics workload and lowest for the MG-RAST workload) (Fig. 8, 10, 11), different (RF,CL) values (larger values of RF and smaller values of CL) (Fig. 10), and data volumes (benefit stays unchanged) (Fig. 8). **Third**, OptimusCloud achieves comparable or better P99 latency than the baselines, thus showing that it does not sacrifice raw performance in search of the performance per unit cost.

MG-RAST			BUS-Tracking		
MC-Order	Lookahead	RMSE	MC-Order	Lookahead	RMSE
First	5m	43.7%	First	15m	6.9%
First	10m	68.7%	**First**	**1h**	**7.4%**
Second	**5m**	**43.4%**	Second	5m	7.12%
Second	10m	68.2%	Second	1h	7.4%

Table 2: (MC stands for Markov Chain). Workload prediction RMSE for MG-RAST and Bus-tracking workloads with different lookahead periods.

4.1 Applications

Here we give the details for our three use case applications, which together span a wide range in terms of predictability and nature of the requests in the workload. **MG-RAST** is a global-scale metagenomics portal [7], the largest of its kind, which allows many users to simultaneously upload their metagenomic data to the repository, apply a pipeline of computationally intensive processes and optionally commit the results back to the repository for shared use. Its workload does not have any discernible daily or weekly pattern, as the requests come from all across the globe and we find that the workload can change drastically over a few minutes. A total of 80 days of real query trace were analyzed, 60 days for training and 20 days for testing. In production, MG-RAST is executed with the values RF=3, CL=1 and it shows abrupt switches in the Read-Ratio (typically from RR=0 to RR=1) and vice versa. The frequency of these switches are 430/day on median. This presents a challenging use case as only 5 minutes of accurate lookahead is possible.

The second workload is the **Bus-Tracking** application [39] where read, scan, insert, and update operations are submitted to a backend database. The data has strong daily and weekly patterns to it. This workload has less frequent switches of 60/day on median. For this workload, 60 days of real query trace were analyzed for the application (40 for training and 20 for testing). The relative proportions of the different kinds of queries are 42.2% updates, 54.8% scans, 2.82% inserts, and 0.18% reads. As shown in Table 2, the prediction accuracy for Bus-Tracking workload is much better compared to the MG-RAST workload, as expected due to the more regular patterns, and here we use a longer lookahead period of 1 hour.

The third use case is a queue of **data analytics jobs** such as would be submitted to an HPC computing cluster. Here the workload can be predicted over long time horizons (order of an hour) by observing the jobs in the queue and leveraging the fact that a significant fraction of the job patterns are recurring. Thus, our evaluation cases span the range of patterns and corresponding predictability of the workloads. We simulate a shared queue of batch data analytics jobs. We modeled the jobs on data analytics requests submitted to a real Microsoft Cosmos cluster [26]. Each job is divided into stages and the workload characteristics of the job change with every stage. The job size is a random variable $\sim$ U(200,100K) operations. The workload switches are 780/day on median, with a level of concurrency of 10 jobs. We achieve accurate prediction over a lookahead duration of 1 hour and we use that for our setting with this use case.

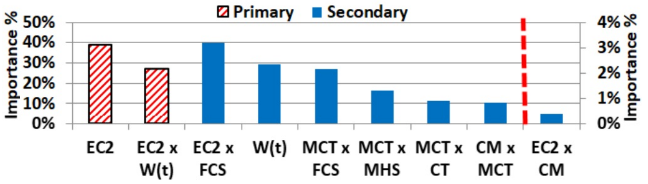


Figure 7: Importance of various parameters, including pairwise combinations. Parameters with black solid bars are w.r.t. the right Y-axis. EC2 configuration, the workload, and top 5 Cassandra parameters describe 81% of data variance, after which there is a significant drop in importance, denoted by the red dotted line. Top parameters are: file_cache_size (FCS), memtable_cleanup_threshold (MCT), memtable_heap_space (MHS), compaction_throughput (CT), and compaction_method (CM)

4.2 Baselines

We compare OPTIMUSCLOUD to the following baselines:

1. *Homogeneous-Static*: We use our cluster predictor to select the single best configuration to use for the entire duration of the predicted workload. The entire workload is assumed to be known in advance, making this an impractically optimistic baseline. Nevertheless, it is a measure of how well a *statically determined* homogeneous configuration can perform when powered by a hypothetically perfect workload predictor.
2. *CherryPick*: We use CherryPick's Bayesian Optimization (BO) to find a heterogeneous cloud configuration which maximize our objective metric. When the workload changes, BO collects 20 points and selects the best cloud configuration. This process takes about 3 minutes with parallelization on a 16-core machine. The reconfiguration is done by restarting servers all at once, thus making data unavailable transiently.
3. *Selecta*: We use Selecta's SVD prediction model to select the optimized homogeneous configuration with workload changes. We populate the SVD matrix with the same training data as used for training of OPTIMUSCLOUD. Further, we give a benefit to Selecta that all the workload characteristics are assumed to be pre-filled in the matrix (or close to it) so that it does not have to execute and profile the workload that arrives at runtime, but can be looked up in the matrix. Both CherryPick and Selecta run reactively with workload changes and neither can change the application configuration. They operate in a greedy manner initiating reconfiguration whenever the workload changes, unlike OPTIMUSCLOUD that optimizes for the workload over a lookahead time window.
4. *SOPHIA*: We use SOPHIA for homogeneous NoSQL configurations while the VM configurations are fixed to the recommended VM types in Cassandra's and Redis' documentations i.e., Compute-Optimize C4.large for Cassandra [2] and Memory-Optimized R4.large for Redis [1].
5. *Theoretical-Best*: This is a baseline that knows what is the best-performing configuration for every workload. It then switches the cluster to this configuration without any downtime cost. Though impractical, this baseline provides a quantitative upper bound for any protocol and is used for normalization of our results.

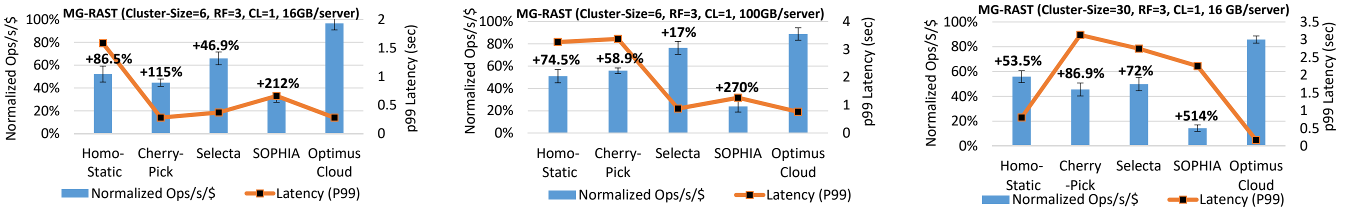


Figure 8: Evaluation of MG-RAST traces in Cassandra using OPTIMUSCLOUD *vs state-of-the-art tuning systems. The primary Y-axis represents the ratio of the normalized Ops/s/$ achieved by each system to the theoretical-best performance.* OPTIMUSCLOUD *achieves the highest Perf/$ and lowest P99 latency.*

4.3 End-to-end System Evaluation

We evaluate how effective OPTIMUSCLOUD and each of the baselines are in selecting the best reconfiguration plan. In Fig. 8 we show the evaluation for the MG-RAST application, the most challenging one for us due to its unpredictable workload characteristics. We use a 1 hour trace from MG-RAST and apply it to a cluster of 6 or 30 servers We show the performance of the different plans with data volume per server of 16GB and 100GB. The performance of each solution is normalized by that of the Theoretical-Best. OPTIMUSCLOUD's plan achieves the highest Ops/s/$ and the lowest latency over all baselines. OPTIMUSCLOUD achieves 86% and 74% improvement over Homogeneous-Static for the 16GB and 100GB cases respectively. This shows there is no single static configuration that can achieve the optimal performance for all phases of the workload. Compared to CherryPick and Selecta, OPTIMUSCLOUD achieves 87% and 45% improvement on average. This is because both systems are striving to create a homogeneous configuration to meet the performance requirement and end up increasing the cost. Further, CherryPick incurs the delay of performing the Bayesian optimization, which takes 3 minutes on average and causes the cluster to operate with sub-optimal configurations during this long delay. The aggressive reconfiguration of CherryPick and Selecta (shutting down all servers and restarting all together) causes a significant performance hit (throughput of zero) for the cluster. Compared to SOPHIA, OPTIMUSCLOUD achieves 212% and 270% improvement in Perf/$. This highlights the benefit of jointly tuning VM and database configurations.

We draw several other conclusions. First, with increasing data volume, the throughput of all protocols goes down because what would once fit in memory (R4.large has 16 GB) now has to go to disk. But the effect on Selecta and CherryPick is smaller because they use expensive and well-resourced memory VMs. Consequently, the performance benefit of OPTIMUSCLOUD decreases. Second, Selecta is achieving better performance than CherryPick, which is consistent with the results reported in [32]. This is because of the longer response time of CherryPick's Bayesian Optimization versus Selecta's matrix lookup. In terms of latency, OPTIMUSCLOUD scales well (comparing the N=6 to N=30), while Selecta and CherryPick both suffer (latency increases of 7.4× and 11×). This is because the control message traffic is very large in these two baselines as they aggressively shut down and restart all the servers together to achieve a reconfiguration, causing the scalability bottleneck. We also notice that SOPHIA scalability is better than the other baselines as it performs sequential reconfiguration. OPTIMUSCLOUD achieves as low tail-latency as Selecta and CherryPick for small scales, and lower at larger scale due to the reason above. Homogeneous-Static has a high latency for all cases due to its inability to adapt to dynamic workloads. The comparison with Selecta and CherryPick shows how important it is to apply *online* reconfiguration to minimize tail-latencies for fast changing workloads.

4.4 Sensitive Parameter Identification

We test the feasibility of pruning the joint configuration search space (i.e., VM and NoSQL), while maintaining the dependencies among the configuration parameters. We collect a total of 3K data points equally from 15 different VM types. We use D-optimal design to decide which data points to collect for building the performance prediction model. Fig. 7 shows the importance of the most impactful parameters, either singly or pairwise, as determined from the regression model. The instance architecture (EC2) and the workload (W(t)) are the most impactful, followed by top-5 database configuration parameters. Note that the costs of changing different configuration parameters are different as it takes about 90 sec to change EC2 type, whereas changing Cassandra's configuration only requires around 30 sec with no impact on the cluster's $ cost. Expectedly, the instance architecture has high inter-dependency with the NoSQL parameters because the architecture controls the physical resources available to the DBMS.

4.5 Single Server Performance Prediction

We evaluate three possible single server prediction models for inclusion in OPTIMUSCLOUD. In each case, we use a Random Forest using 75%:25% for training and prediction.
1. *N-Solitary-Models*: This builds a separate prediction model per architecture. It predicts the performance of a given architecture/configuration combination using previously collected data points from the same architecture.
2. *Combined-Categorical*: This builds a combined model using all points from all architectures, while it represents the architecture as a categorical parameter (with integral values).

Workload	MG-RAST		BUS		HPC	
Metric	R^2	*RMSE*	R^2	*RMSE*	R^2	*RMSE*
N-Solitary -Models	0.2	3401.4	0.127	109.9	0.04	2778
Selecta	-0.14	4149.3	0.66	110.6	0.932	2451
OPTIMUS -Categorical	0.41	1334.2	0.986	21.87	0.983	1172.9
OPTIMUS -Numerical	**0.89**	**1260.9**	**0.988**	**19.77**	**0.986**	**1076.2**

Table 3: ***Comparison of different Single-server prediction techniques.*** OPTIMUSCLOUD ***achieves better performance in terms of R^2 and RMSE over all baselines.***

Thus, knowledge transfer is limited across architectures.
3. *Combined-Numerical*: This also builds a combined model for all architectures. However, it describes the architecture in terms of its resources e.g., C4.large is represented as vCPU 8, RAM 3.75 GB, Network-Bandwidth 0.62 Gbits/s. Thus this allows extrapolating model knowledge across architectures. We test the accuracy of each predictor using the same number of data points (100 points per architecture) and show the result in terms of R^2 (Table 3). We see that using a separate model per architecture gives very poor performance due to the lack of knowledge transfer between architectures. Moreover, the numerical representation shows a significant improvement in prediction performance over the categorical representation due to better knowledge transfer across architectures. Thus, we use the *Combined-Numerical* model in OPTIMUSCLOUD.

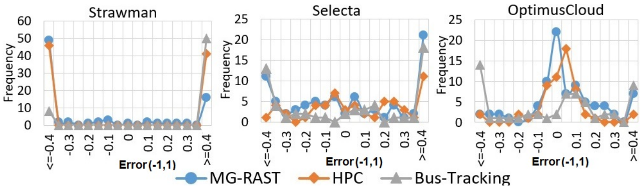


Figure 9: Performance prediction error histogram for heterogeneous clusters. We notice that OPTIMUSCLOUD***'s error percentage is within -15% to +15% for 70% of the test points, with R^2 value of 0.91 and RMSE of 7.7 KOps/s. On the other hand, the best strawman shows poor performance (RMSE 36 KOps/s) while Selecta performs better (RMSE 21 KOps/s).***

4.6 Cluster Performance Prediction

We evaluate the accuracy of our cluster performance prediction model. We use a cluster of 6 nodes with RF=3, CL=1 and investigate the impact of changing the EC2 architecture of each *Complete-Set*. Thus, the cluster is partitioned into 3 *Complete-Set*s. We use 3 families of EC2's 4th generation (C4, R4, M4) and three different sizes of each family (large, xlarge, 2xlarge). We collect 330 data points covering all combinations of assigning instance types to these 3 *Complete-Set*s.

In Fig. 9, we compare our model with a strawman predictor that uses the sum of Ops/s for each individual server as the overall cluster performance (we also tested Average, Min, and Max and got worse performance). This strawman achieves poor prediction performance with a low R^2 value of 0.08 and an unacceptably high RMSE of 36 KOps/s. We also compare our model with the latent factor collaborative filtering technique, SVD, used in Selecta [32], which we reimplement using the sci-kit `surprise` library [30]. Selecta achieved better R^2 value of 0.69 and a lower RMSE of 21 KOps/s compared to the strawman predictor. However, our model achieves better performance due to the fact that our Random-Forest model can use non-linear combinations of the elementary features (up to quadratic), while SVD is confined to using linear combinations only. The shapes of the error curves are also different—the Selecta and the strawman curves are bathtub-shaped indicating significant overestimation or underestimation, while the OPTIMUSCLOUD curve is bell-shaped with a mean close to zero. The bathtub curves are due to the fact that these protocols are ignorant of the token assignment of the cluster and consider erroneously that each node's throughput has the same contribution to the cluster throughput.

4.7 Evaluation with Diverse Workloads

Now we evaluate the performance for different workloads, cluster scales, and (RF, CL) requirements.

HPC Workload: Figure 10 shows the improvement of OPTIMUSCLOUD over the baselines for the HPC data analytics traces. We first change RF from 1 to 3, holding CL at 1. Then we change CL from 1 to quorum, which is 2, holding RF at 3. OPTIMUSCLOUD's plan achieves the highest Perf/$ and the lowest latency over all baselines for all setups. At RF=3, OPTIMUSCLOUD achieves 20% and 24% better Perf/$ over Homogeneous-Static configuration for CL=1 and CL=Q respectively. This again shows the importance of dynamic reconfiguration to handle workloads with changing characteristics. In comparison to CherryPick, OPTIMUSCLOUD achieves 143% and 89% better performance for CL=1 and CL=Q respectively and 130% and 80% over Selecta. Compared to SOPHIA, OPTIMUSCLOUD achieves 23% and 12.5% better Perf/$. Notice that when RF=1 and CL=1, OPTIMUSCLOUD can no longer perform non-atomic configurations since the *Complete-Set* in this case is the entire cluster. Thus, its improvement over Homogeneous-Static decreases to 9%.
As RF goes up, the amount of data on each node goes up bringing down the absolute performance. Homogeneous-Static is affected more than OPTIMUSCLOUD and therefore the benefit of OPTIMUSCLOUD increases. CherryPick and Selecta are relatively unaffected by increasing RF as they had low throughputs to begin with and they reconfigure all the nodes at once, irrespective of RF and CL. SOPHIA benefits from increasing RF since it can reconfigure more servers concurrently without degrading data availability. As CL goes up, again CherryPick and Selecta are relatively unaffected. The performance of OPTIMUSCLOUD goes down because the maximum number of *Complete-Sets* that OPTIMUSCLOUD can reconfigure at a time is inversely proportional to CL. Thus, its benefit over CherryPick and Selecta reduces. However, we note that for most of our target environments, CL is unlikely

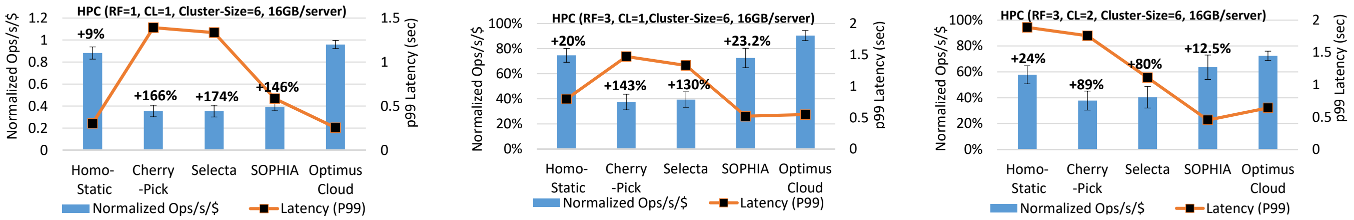


Figure 10: HPC workload evaluation with 10 concurrent jobs, and varying (RF,CL) requirements

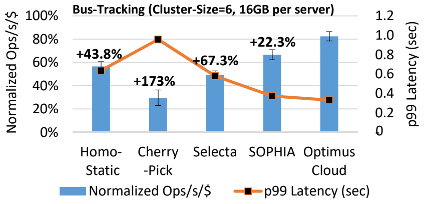


Figure 11: Bus-Tracking workload pattern with RF=3, CL=1

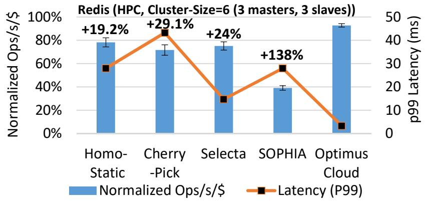


Figure 12: Evaluation on Redis for HPC workload with a cluster of 6 servers

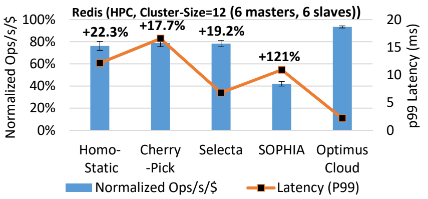


Figure 13: Evaluation on Redis for HPC workload with a cluster of 12 servers

to be higher than 1 as deployments often favor availability and latency over consistency. In terms of latency, OPTIMUSCLOUD achieves the lowest P99 latency across all setups. We also notice that SOPHIA has lower latency than other baselines as it performs a gradual reconfiguration of the different server instances to maintain data availability.

Bus-Tracking Workload: This workload has strong weekly and daily patterns, which allows the workload predictor to provide accurate predictions for long lookahead periods. For a 1 day-trace, the result is shown in Fig. 11. OPTIMUSCLOUD achieves better performance/$ over Homogeneous-Static, CherryPick, Selecta, and SOPHIA by 46%, 178%, 67%, and 28% respectively. As before, OPTIMUSCLOUD achieves the lowest tail latency across all techniques. The tail latency metric is very important for this workload as it represents a user-facing application. We observe that OPTIMUSCLOUD achieves higher gains in performance over CherryPick and Selecta compared to the MG-RAST workload, which shows the benefit of longer accurate predictions for OPTIMUSCLOUD.

4.8 Evaluation with Redis

Redis has a very different architecture than Cassandra and is therefore a suitable target to evaluate the generalizability of OPTIMUSCLOUD. Here we use Redis in clustered mode as a distributed cache (a common use case for Redis)—if the key is found in Redis' memory, it is served by Redis, else, it is served by a slower disk-based database. We apply the HPC analytics workload to a cluster of 6 or 12 Redis servers, keeping the replication degree as 2. We select HPC workload as it has the shortest key-reuse-distance between the three workloads and for which using Redis as a cache is most beneficial [10, 21]. We tune Redis' `maxmemory-policy` and `maxmemory` parameters and observe that changing the cloud configurations for more or less RAM size has an impact on the best value of both parameters. We also found that Redis' Perf/$ is sensitive to workload parameters such as job size, access distribution, and read-to-write. We start by collecting 108 data points for different jobs with varying job sizes, access distributions, and read-write ratios. Job size is a random variable with U(0.5M,1.5M) operations. Access distribution is randomly selected from *Uniform*, *Latest*, and *Zipfian*. Read-write ratio is a random variable with distribution U(0,1). We experiment with traces of 75 jobs which span a total of 5 hours. Our performance predictor achieves an R^2 of 0.922 averaged over 20 runs. From Fig. 12, we see that OPTIMUSCLOUD achieves a better Perf/$ of 19% (Homogeneous-Static), 29% (CherryPick), 24% (Selecta), and 138% (SOPHIA). Also, OPTIMUSCLOUD reduces the tail latency by 8.4X (Homogeneous-Static), 13X (CherryPick), 4.4X (Selecta), and 8.1X (SOPHIA).

We draw the following insights. First, as SOPHIA uses an expensive cluster of R4.large (as recommended in Redis' documentation [1]), it achieves a very low Perf/$. Second, both CherryPick and Selecta switch from R4.large to the less expensive C4.large and M4.large when the workload changes (and less memory is required) and therefore achieve a higher Perf/$. Finally, by using a heterogeneous cluster of the three VM types as well as jointly tuning the application configuration, we achieve the best Perf/$ and the lowest latency among all techniques. To test scalability, we increase the number of servers to 12 (Fig. 13) and note that the normalized performance of each system stays approximately constant.

4.9 Tolerance to Prediction Errors

We investigate how tolerant OPTIMUSCLOUD is to errors in both predictors—performance (throughput) and workload. We add synthetic noise to the output of each predictor separately and then show how does the benefit of OPTIMUSCLOUD change with the amount of synthetic noise for the HPC workload. The percentage of noise is represented as a uniform random variable that is added to (or subtracted from) the output of the predictor. For performance prediction,

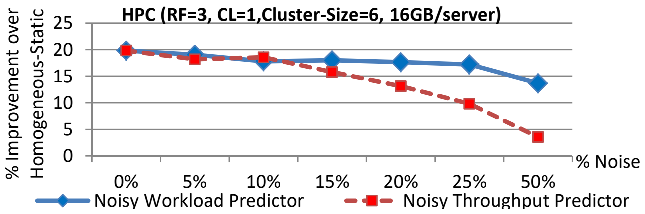


Figure 14: Impact of noisy predictions on* OptimusCloud*'s improvement over best Homogeneous-Static configurations

the noise is added to the overall throughput/$ predicted by our multi-server model. For workload prediction, the noise is added to the number of requests/sec in addition to the workload change duration. As shown in Fig.14, OptimusCloud's improvement over Homogeneous-Static decreases with increasing levels of noise, as the selected configurations deviate from the best configurations. We note that OptimusCloud is more sensitive to errors in the throughput predictor compared to errors in the workload predictor, which is demonstrated in the steeper downward slope in the noisy throughput predictor curve. The reason for this high sensitivity is that OptimusCloud uses the throughput predictor to select the best configuration and with increasing levels of noise, the selected configuration more frequently deviates from the optimal. As discussed earlier, a slight deviation from the optimal configuration may cause a significant reduction in Perf/$. On the other hand, slight errors in workload prediction causes OptimusCloud to reconfigure earlier or later than it optimally should. However, this has less impact on performance as long as it still switches to the best configuration.

5 Related Work

Configuration tuning for datastores: Several works [6, 31] target online configuration tuning for replicated or geo-replicated datastores. [27, 56] perform online reconfiguration for NoSQL datastores. None of these works address how to optimize for cost-performance benefits by exploiting different cloud VM/instance types. A large body of work focused on the best logical or physical design for static workloads in DBMS [3,12,13,18,29,51,52,61], which are orthogonal to our work. OtterTune [62], BerkeleyDB [60], and iTuned [24] only optimize DBMS configuration, while OptimusCloud optimizes both NoSQL configuration and the cluster on which it runs. Pocket [33] optimizes the storage servers for ephemeral data analytics jobs in contrast to handling long-running jobs that are OptimusCloud's focus. While OptimusCloud can also optimize storage, we choose to restrict our storage servers to elastic block storage (EBS) that are separate from the VMs and thus retain data durability.

Performance predictions: Ernest [63] predicts performance of data analytics applications through system modeling. DBSeer [46] uses linear models to predict resource utilization (e.g. Disk I/O). Myria [49, 66] gives personalized SLAs by predicting query execution times for specific workloads. Recent works [42, 43, 45] improve utilization by *learning* workload characteristics. [44] handles prediction errors for unseen workloads. [23] uses queuing models. Rafiki [40] uses a surrogate model to predict performance of NoSQL datastores. No prior work predicts performance for heterogeneous clusters.

Cache Hit-Rate Maximization: Several works target maximizing cache hit-rates and improving end-to-end latency, either for a single application [14] or multi-tenant deployments [15]. However, neither VM configurations nor heterogeneous cluster configurations are considered to optimize performance/$. [9, 65] propose new cache eviction policies that can be implemented in Redis and then selected by OptimusCloud for the appropriate workloads.

6 Discussion

Impact on consistency: OptimusCloud exploits the fact that in typical NoSQL deployments, RF>CL as availability and low-latency are favored over consistency [22, 47, 57]. The higher the difference between RF and CL, the smaller the subset of servers that needs reconfiguration, therefore the higher the gain of OptimusCloud over baselines (Figure 10). For users whose primary goal is consistency and want to use a high value for CL, one option is to also increase RF and leverage the high gain of OptimusCloud. However, increasing RF also increases the total number of copies stored in the cluster, which might negatively impact the cluster's write performance in exchange for higher availability.

Compatibility with other key-value stores: OptimusCloud's design assumes that the underlying key-value store implements a protocol to identify and select the fastest replica(s) given a new query. Cassandra achieves this using its dynamic snitching policy, while other popular systems have similar protocols (e.g., Elasticsearch achieves this by its Adaptive Replica Selection policy [25]). If this feature is not implemented in the system, a simple solution is for OptimusCloud to provide the system with the ordered list of replicas, using OptimusCloud 's performance predictor.

7 Conclusion

For cost-optimal performance of a distributed NoSQL cloud database, it is critical to jointly tune NoSQL and cloud configurations. OptimusCloud provides the insight that it is optimal to create heterogeneous configurations and for this, it determines at runtime the *minimum number of servers* to reconfigure. Using a novel concept of Complete Sets, OptimusCloud provides a technique to search through the large search space brought out by heterogeneity. Configurations found by OptimusCloud outperform those by prior works, CherryPick, Selecta, and SOPHIA, in both Perf/$ and tail latency, across two NoSQL DBMSs, Cassandra and Redis, and all experimental conditions.

References

[1] 5 tips for running redis over aws. https://redislabs.com/blog/5-tips-for-running-redis-over-aws/. [Online; accessed 17-September-2019].

[2] Best practices for running apache cassandra on amazon ec2. https://aws.amazon.com/blogs/big-data/best-practices-for-running-apache-cassandra\-on-amazon-ec2/. [Online; accessed 17-September-2019].

[3] AGRAWAL, S., NARASAYYA, V., AND YANG, B. Integrating vertical and horizontal partitioning into automated physical database design. In *ACM SIGMOD international conference on Management of data* (2004).

[4] AKDERE, M., ÇETINTEMEL, U., RIONDATO, M., UPFAL, E., AND ZDONIK, S. B. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering* (2012), IEEE, pp. 390–401.

[5] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation NSDI'17* (2017), pp. 469–482.

[6] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *OSDI* (2014), pp. 367–381.

[7] ARGONNE NATIONAL LABORATORY. MG-RAST: Metagenomics Analysis Server. urlhttps://www.mg-rast.org/, 2019.

[8] AWS. Amazon EC2. https://aws.amazon.com/ec2/pricing/on-demand/, May 2018.

[9] BECKMANN, N., CHEN, H., AND CIDON, A. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation NSDI'18* (2018), pp. 389–403.

[10] BEYLS, K., AND D'HOLLANDER, E. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems* (2001), vol. 14, pp. 350–360.

[11] CHATERJI, S., KOO, J., LI, N., MEYER, F., GRAMA, A., AND BAGCHI, S. Federation in genomics pipelines: techniques and challenges. *Briefings in bioinformatics 20*, 1 (2019), 235–244.

[12] CHAUDHURI, S., AND NARASAYYA, V. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 3–14.

[13] CHAUDHURI, S., AND NARASAYYA, V. R. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB* (1997).

[14] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation NSDI'16* (2016), pp. 379–392.

[15] CIDON, A., RUSHTON, D., RUMBLE, S. M., AND STUTSMAN, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference ATC'17* (2017), pp. 321–334.

[16] CLUTCH. Data replication in cassandra. https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html, 2019.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[18] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *VLDB Endowment* (2010).

[19] DATASTAX. Cassandra dynamic snitching. https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archSnitchDynamic.html, 2019.

[20] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[21] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices* (2003), vol. 38, ACM, pp. 245–257.

[22] DIOGO, M., CABRAL, B., AND BERNARDINO, J. Consistency models of nosql databases. *Future Internet 11*, 2 (2019), 43.

[23] DIPIETRO, S., CASALE, G., AND SERAZZI, G. A queueing network model for performance prediction of apache cassandra. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools* (2017), pp. 186–193.

[24] DUAN, S., THUMMALA, V., AND BABU, S. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment 2*, 1 (2009), 1246–1257.

[25] ELASTICSEARCH. Improving Response Latency in Elasticsearch with Adaptive Replica Selection. urlhttps://www.elastic.co/blog/improving-response-latency-in-elasticsearch-with-adaptive-replica-selection, 2020.

[26] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings*

of the 7th ACM european conference on Computer Systems (2012), ACM, pp. 99–112.

[27] GHOSH, M., WANG, W., HOLLA, G., AND GUPTA, I. Morphus: Supporting online reconfigurations in sharded nosql systems. *IEEE Transactions on Emerging Topics in Computing* (2015).

[28] GHOSHAL, A., GRAMA, A., BAGCHI, S., AND CHATERJI, S. An ensemble svm model for the accurate prediction of non-canonical microrna targets. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics* (2015), pp. 403–412.

[29] GUPTA, H., HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. D. Index selection for olap. In *IEEE International Conference on Data Engineering (ICDE)* (1997).

[30] HUG, N. Surprise, a python library for recommender systems. `http://surpriselib.com`, 2017.

[31] KEMME, B., BARTOLI, A., AND BABAOGLU, O. Online reconfiguration in replicated databases based on group communication. In *Dependable Systems and Network (DSN)* (2001), IEEE, pp. 117–126.

[32] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Selecta: heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference ATC'18* (2018), pp. 759–773.

[33] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018), pp. 427–444.

[34] KLOPHAUS, R. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming* (2010), ACM, p. 14.

[35] KONSTANTINOU, I., TSOUMAKOS, D., MYTILINIS, I., AND KOZIRIS, N. Dbalancer: distributed load balancing for nosql data-stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1037–1040.

[36] KOO, J., ZHANG, J., AND CHATERJI, S. Tiresias: Context-sensitive approach to decipher the presence and strength of microrna regulatory interactions. *Theranostics 8*, 1 (2018), 277.

[37] KOUSIOURIS, G., CUCINOTTA, T., AND VARVARIGOU, T. The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks. *Journal of Systems and Software 84*, 8 (2011), 1270–1291.

[38] KOZIEL, S., AND YANG, X.-S. *Computational optimization, methods and algorithms*, vol. 356. Springer, 2011.

[39] MA, L., VAN AKEN, D., HEFNY, A., MEZERHANE, G., PAVLO, A., AND GORDON, G. J. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data* (2018), ACM, pp. 631–645.

[40] MAHGOUB, A., WOOD, P., GANESH, S., MITRA, S., GERLACH, W., HARRISON, T., MEYER, F., GRAMA, A., BAGCHI, S., AND CHATERJI, S. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 28–40.

[41] MAHGOUB, A., WOOD, P., MEDOFF, A., MITRA, S., MEYER, F., CHATERJI, S., AND BAGCHI, S. SOPHIA: Online reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference ATC'19* (2019), Usenix, pp. 223–240.

[42] MAO, H., ALIZADEH, M., MENACHE, I., AND KANDULA, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), pp. 50–56.

[43] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., MENG, Z., AND ALIZADEH, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[44] MITRA, S., BRONEVETSKY, G., JAVAGAL, S., AND BAGCHI, S. Dealing with the unknown: Resilience to prediction errors. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), IEEE, pp. 331–342.

[45] MITRA, S., MONDAL, S. S., SHEORAN, N., DHAKE, N., NEHRA, R., AND SIMHA, R. Deepplace: Learning to place applications in multi-tenant clusters. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (2019), pp. 61–68.

[46] MOZAFARI, B., CURINO, C., AND MADDEN, S. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR* (2013).

[47] NEGRIN, R. Thank you for your help nosql, but we got it from here. https://www.memsql.com/blog/why-nosql-databases-wrong-tool-for-modern-application/, 2018.

[48] OF STANDARDS, N. I., AND (NIST), T. D-optimal designs. `https://itl.nist.gov/div898/handbook/pri/section5/pri521.htm`, 2019.

[49] ORTIZ, J., DE ALMEIDA, V. T., AND BALAZINSKA, M. Changing the face of database cloud services with personalized service level agreements. In *CIDR* (2015).

[50] PALCZEWSKA, A., PALCZEWSKI, J., ROBINSON, R. M., AND NEAGU, D. Interpreting random forest classification models using a feature contribution

method. In *Integration of reusable systems*. Springer, 2014, pp. 193–218.

[51] PAVLO, A., JONES, E. P., AND ZDONIK, S. On predictive modeling for optimizing transaction execution in parallel oltp systems. *VLDB Endowment* (2011).

[52] RAO, J., ZHANG, C., MEGIDDO, N., AND LOHMAN, G. Automating physical database design in a parallel database. In *ACM SIGMOD international conference on Management of data* (2002).

[53] REDIS. Redis cluster tutorial. `https://redis.io/topics/cluster-tutoriallpp`, February 2015.

[54] REDIS. Using redis as an lru cache. `https://redis.io/topics/lru-cache`, 2019.

[55] SCIKITLEARN. scikit-learn: Machine learning in python. `https://scikit-learn.org/stable/`, 2019.

[56] SHIN, Y., GHOSH, M., AND GUPTA, I. Parqua: Online reconfigurations in virtual ring-based nosql systems. In *2015 International Conference on Cloud and Autonomic Computing* (2015), IEEE, pp. 220–223.

[57] SINGH, V. K. Sql vs nosql databases. https://medium.com/system-design-blog/sql-vs-nosql-databases-6896a8cb1800, 2019.

[58] SOLID. Solid:a comprehensive gradient-free optimization framework written in python. `https://github.com/100/Solid`, 2019.

[59] SRINIVAS, M., AND PATNAIK, L. M. Genetic algorithms: A survey. *computer 27*, 6 (1994), 17–26.

[60] SULLIVAN, D. G., SELTZER, M. I., AND PFEFFER, A. *Using probabilistic reasoning to automate software tuning*, vol. 32. ACM, 2004.

[61] VALENTIN, G., ZULIANI, M., ZILIO, D. C., LOHMAN, G., AND SKELLEY, A. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *IEEE International Conference on Data Engineering (ICDE)* (2000).

[62] VAN AKEN, D., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1009–1024.

[63] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation NSDI'16* (2016), pp. 363–378.

[64] W, G. F., AND A, K. G. *Handbook of metaheuristics*, vol. 57. Springer Science & Business Media, 2006.

[65] WALDSPURGER, C., SAEMUNDSSON, T., AHMAD, I., AND PARK, N. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference ATC'17* (2017), pp. 487–498.

[66] WANG, J., BAKER, T., BALAZINSKA, M., HALPERIN, D., HAYNES, B., HOWE, B., HUTCHISON, D., JAIN, S., MAAS, R., MEHTA, P., ET AL. The myria big data management and analytics system and cloud services. In *CIDR* (2017).

[67] YADWADKAR, N. J., HARIHARAN, B., GONZALEZ, J. E., SMITH, B., AND KATZ, R. H. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 452–465.

Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum,
Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini *

Microsoft Azure and Microsoft Research

Abstract

Function as a Service (FaaS) has been gaining popularity as a way to deploy computations to serverless backends in the cloud. This paradigm shifts the complexity of allocating and provisioning resources to the cloud provider, which has to provide the illusion of always-available resources (*i.e.*, fast function invocations without cold starts) at the lowest possible resource cost. Doing so requires the provider to deeply understand the characteristics of the FaaS workload. Unfortunately, there has been little to no public information on these characteristics. Thus, in this paper, we first characterize the entire production FaaS workload of Azure Functions. We show for example that most functions are invoked very infrequently, but there is an 8-order-of-magnitude range of invocation frequencies. Using observations from our characterization, we then propose a practical resource management policy that significantly reduces the number of function cold starts, while spending fewer resources than state-of-the-practice policies.

1 Introduction

Function as a Service (FaaS) is a software paradigm that is becoming increasingly popular. Multiple cloud providers offer FaaS [5, 17, 21, 28] as the interface to usage-driven, stateless (serverless) backend services. FaaS offers an intuitive, event-based interface for developing cloud-based applications. In contrast with the traditional cloud interface, in FaaS, users do not explicitly provision or configure virtual machines (VMs) or containers. FaaS users do not pay for resources they do not use either. Instead, users simply upload the code of their functions to the cloud; functions get executed when "triggered" or "invoked" by events, such as the receipt of a message (*e.g.*, an HTTP request) or a timer going off. The provider is then responsible for provisioning the needed resources (*e.g.*, a container in which to execute each function), providing high function performance, and billing users just for their actual function executions (*e.g.*, in increments of 100 milliseconds).

Obviously, providers seek to achieve high function performance at the lowest possible resource cost. There are three main aspects to how fast functions can execute and the resources they consume. First, function execution requires having the needed code (*e.g.*, user code, language runtime libraries) in memory. A function can be started quickly when the code is already in memory (warm start) and does not have to be brought in from persistent storage (cold start). Second, keeping the resources required by all functions in memory at all times may be prohibitively expensive for the provider, especially if function executions are short and infrequent. Ideally, the provider wants to give the illusion that all functions are always warm, while spending resources as if they were always cold. Third, functions may have widely varying resource needs and invocation frequencies from multiple triggers. These characteristics severely complicate any attempts to predict invocations for reducing resource usage. For example, the wide range of invocation frequencies suggests that keeping resources in memory may work well for some functions but not others. With respect to triggers, HTTP triggers may produce invocations at irregular intervals that are difficult to predict, whereas timers are regular.

These observations make it clear that providing high function performance at low cost requires a deep understanding of the characteristics of the FaaS workload. Unfortunately, there has been no public information on the characteristics of production workloads. Prior work [3, 15, 24, 25, 27, 44] has focused on either (1) running benchmark functions to assess performance and/or reverse-engineer how providers manage resources; or (2) implementing prototype systems to run benchmark functions. In contrast, what is needed is a comprehensive characterization of the users' *real* FaaS workloads on a *production* platform from the provider's perspective.

Characterizing production workloads. To fill this gap, in this paper, we first characterize the entire production FaaS workload of Azure Functions [28]. We characterize the real functions and their trigger types, invocation frequencies and patterns, and resource needs. The characterization produces many interesting observations. For example, it shows that most functions are invoked very infrequently, but the most popular functions are invoked 8 orders of magnitude more frequently than the least popular ones. It also shows that functions exhibit a variety of triggers, producing invocation patterns that are often difficult to predict. In terms of resource needs, the characterization shows a 4x range of function memory usage and that 50% of functions run in less than 1 second.

Researchers can use the distributions of the workload characteristics we study to create realistic traces for their work.

[*] Shahrad is affiliated with Princeton University, but was at MSR during this work. Laureano and Tresness are now with Facebook and D. E. Shaw.

Alternatively, they can use the sanitized *production traces* we are making available with this paper [31].

Managing cold-start invocations. Using observations from our characterization, we also propose a practical resource management policy for reducing the number of cold start executions while consuming no more resources than the large cloud providers' current policies. Specifically, AWS and Azure use a fixed "keep-alive" policy that retains the resources in memory for 10 and 20 minutes after a function execution, respectively [39, 40]. Though this policy is simple and practical, it disregards the functions' actual invocation frequency and patterns, and thus behaves poorly and wastes resources.

In contrast, our policy (1) uses a different keep-alive value for each user's workload, according to its actual invocation frequency and pattern; and (2) enables the provider in many cases to pre-warm a function execution just before its invocation happens (making it a warm start). Our policy leverages a small histogram that keeps track of the recent function inter-invocation times. For workloads that exhibit clear invocation patterns, the histogram makes clear how much keep-alive is beneficial and when the pre-warming should take place. For workloads that do not, our policy reverts back to the fixed keep-alive policy. As the histogram must be small, for any workloads that cannot be captured by the histogram but exhibit predictable invocation patterns, our policy uses time-series analysis to predict when to pre-warm.

We implement our policy in simulation and for the Apache OpenWhisk [34] FaaS platform, both driven with real workload traces. Our simulation results show that the policy significantly reduces the number of function cold starts, while spending fewer resources than the fixed keep-alive policy. Our experimental results show that the policy can be easily implemented in real systems with minimal overheads. In fact, we describe our recent production implementation in Azure Functions in the end of the paper.

Contributions. In summary, our main contributions are:

- A detailed characterization of the entire production FaaS workload at a large cloud provider;
- A new policy for reducing the number of cold start function executions at a low resource provisioning cost;
- Extensive simulation and experimental results based on real traces showing the benefits of the policy;
- An overview of our implementation in Azure Functions;
- A large sanitized dataset containing production FaaS traces.

2 Background

Abstraction. In FaaS, the user uploads code to the cloud, and the provider enables a handle (*e.g.*, a URL) for the code to be run. The choices of which resources to allocate, when to allocate them, and for how long to retain them, still have to be made, but they are shifted to the cloud provider.

Triggers. Functions can be invoked in response to several event types, called triggers [6, 29]. For clarity, in this paper we group Azure's many triggers into 7 classes: HTTP, Event, Queue, Timer, Orchestration, Storage, and others. Event triggers include Azure Event Hub and Azure Event Grid, and are used for discrete or serial events, with individual or batch processing. Queue-triggered functions respond to message insertion in a number of message queueing solutions, such as Azure Service Bus and Kafka. Timer triggers are similar to cron jobs, and cause function invocations at pre-determined, regular intervals. We grouped all triggers related to Azure Durable Functions [30] as Orchestration. One can use these triggers to create native, complex function chaining and orchestration. Finally, we grouped database and filesystem triggers as Storage. These fire in response to changes in the underlying data, and include Azure Blob Storage and Redis.

Applications. In Azure Functions, functions are logically grouped in applications, *i.e.* an application may encompass multiple functions. The application concept helps organize the software and in packaging. *The application, not the function, is the unit of scheduling and resource allocation*.

Cold starts. A *cold start* invocation occurs when a function is triggered, but its application is not yet loaded in memory. When this happens, the platform instantiates a "worker" [1] for the application, loads all the required runtime and libraries, and calls the function. This process can take a long time relative to the function execution [44]. There are strategies to reduce the time taken by each cold start, such as keeping pre-allocated VMs or containers, instantiated virtual network interfaces [32], or pre-loaded runtimes that can be specialized on-demand [18]. In this paper, we focus on the complementary and orthogonal goal of reducing the number of cold starts.

Concurrency and elasticity. A running instance of an application can respond to a configurable number of concurrent invocations of its functions. The number depends on the nature of the function, and its resource needs. Cold starts can also happen if there is a spike in the load to an application, and new instances have to be allocated quickly. Given full-server instances and our real FaaS workload, only a tiny percentage (<1%) of applications would experience this type of cold start. For this reason, we do not consider it in this paper.

Cold start management policy. A key aspect of FaaS is the trade-off between reducing cold starts by keeping instances warm, and the resources (*e.g.*, VMs, memory) they need.

Most FaaS providers use a fixed keep-alive policy for all applications, where application instances are kept loaded in memory for a fixed amount of time after a function execution [39, 40]. This is also the case for most open-source implementations (*e.g.*, OpenWhisk uses a 10-minute period).

This policy is simple to implement and maintain, but does not consider the wide variety of application behaviors our characterization unearths. Thus, it can have many cold starts while wasting resources for many applications. Moreover, it is easy to identify by external users, who sometimes invoke their applications frequently enough (perhaps with dummy

[1] In some systems, a worker is a container, but in others it can be a VM.

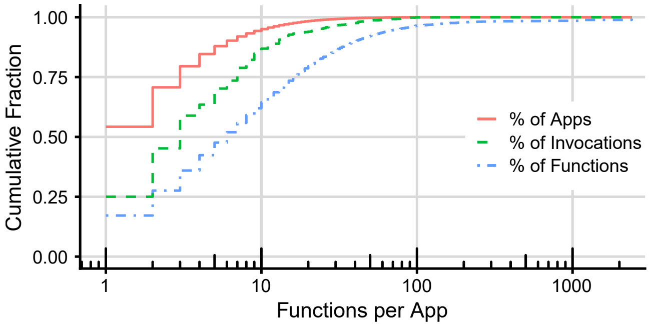


Figure 1: Distribution of the number of functions per app.

invocations) to keep them warm. This practice amplifies the resource waste issue. In this paper, we design a better policy.

3 FaaS Workloads

We characterize the FaaS workloads seen by Azure Functions, focusing on characteristics that are intrinsic to the applications and functions (*e.g.*, their arrival pattern), and not on the characteristics that relate to the underlying platform (*e.g.*, where functions are scheduled). Throughout the characterization, we highlight interesting observations and their implications for cold starts and resource management.

3.1 Data Collection

We collected data on all function invocations across Azure's entire infrastructure between July 15th and July 28th, 2019. We collected four related data sets:

1. Invocation counts: per function, in 1-minute bins;
2. Trigger per function;
3. Execution time per function: average, minimum, maximum, and count of samples, for each 30-second interval, recorded per worker; and
4. Memory usage per application: sampled every 5 seconds by the runtime and averaged, for each worker, each minute. Average, minimum, maximum, and count of samples, for allocated and resident memory.

With this paper, we are releasing a subset of our traces at https://github.com/Azure/AzurePublicDataset.

Limitations. Given the extreme scale of Azure Functions, the invocation counts are binned in 1-minute intervals, *i.e.* our dataset does not allow the precise reconstruction of interarrival times that are smaller than one minute. For this paper, this granularity is sufficient.

For the execution time, we also do not have the complete time distribution across all invocations. However, from the many samples of average time, and corresponding counts, we keep a set of weighted percentiles, where the weight of an entry is the number of samples. For example, if we see an average time of 100ms over 45 samples, the resulting percentiles are equivalent to those computed over a distribution where 100ms are replicated 45 times. The quality of the approximation to the true distribution depends on the number of samples in each bin, the smaller the better. We similarly obtain weighted percentiles for memory usage.

Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

Figure 2: Functions and invocations per trigger type.

For confidentiality reasons, we cannot disclose some absolute numbers, such as total number of functions and invocations. Nevertheless, our characterization is useful for understanding a full FaaS workload, and for researchers and practitioners to generate realistic FaaS workloads.

3.2 Functions, Applications, and Triggers

Functions and applications. Figure 1 shows the CDF of the number of functions per application (top curve). We observe that 54% of the applications only have one function, and 95% of the applications have at most 10 functions. About 0.04% of the applications have more than 100 functions.

The other two curves show the fraction of invocations, and functions, corresponding to applications with up to a certain number of functions. For example, we see that 50% of the invocations come from applications with at most 3 functions, and 50% of the functions are part of applications with at most 6 functions. Though we found a weak positive correlation between the number of functions in an application and the median number of invocations of those applications, the number of functions in an application is not a useful signal in resource management.

We took a closer look at the 10 applications with the most functions. Only 4 had more than 1k functions: these, and 3 others, had a pattern of auto-generated function names triggered by timers or HTTP, which suggests that they were being used for large automated testing. Of the remaining 3 applications, two were using Azure Durable Functions for orchestrating multiple functions, and one seems to be an API application, where each function corresponds to one route in a large Web or REST application. We plan to do a broader and more comprehensive study of application patterns in future work.

Triggers and applications. Figure 2 shows the fraction of all functions, and all invocations, per type of trigger. HTTP is the most popular in both dimensions. Event triggers correspond to only 2.2% of the functions, but to 24.7% of the invocations, due to their automated, and very high, invocation rates. Queue triggers also have proportionally more invocations than functions (33.5% vs 15.2%). The opposite happens with timer triggers. There are many functions triggered by timers (15.6%), but they correspond to only 2% of the invocations, due to the relatively low rate they fire in: 95% of the timer-triggered functions in our dataset were triggered at most once per minute, on average.

Trigger Type	% Apps
HTTP (H)	64.07
Timer (T)	29.15
Queue (Q)	23.70
Storage (S)	6.83
Event (E)	5.79
Orchestration (O)	3.09
Others (o)	6.28

(a) Apps with ≥ 1 of each trigger.

Trigger Types	Fraction of Apps (%)	Cum. Frac. (%)
H	43.27	43.27
T	13.36	56.63
Q	9.47	66.10
HT	4.59	70.69
HQ	4.22	74.92
E	3.01	77.92
S	2.80	80.73
TQ	2.57	83.30
HTQ	2.48	85.78
Ho	1.69	87.48
HS	1.05	88.53
HO	1.03	89.56

(b) Popular trigger combinations.

Figure 3: Trigger types in applications.

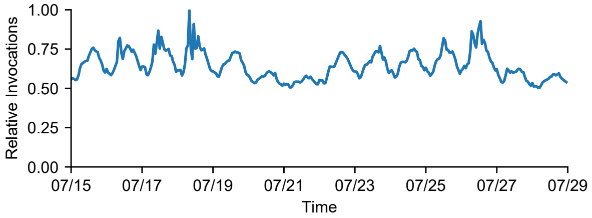


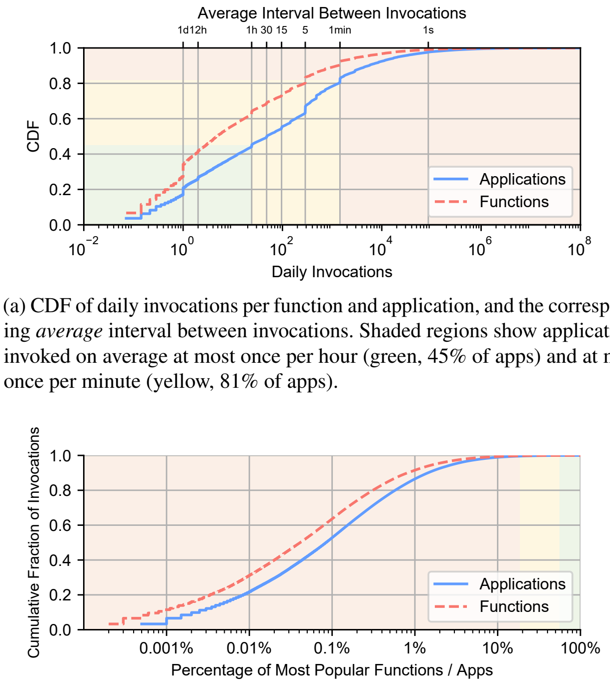
Figure 4: Invocations per hour, normalized to the peak.

Figure 3 shows how applications combine functions with different trigger types. In Figure 3(a), we show the applications with at least one trigger of the given type. We find that 64% of the applications have at least one HTTP trigger, and 29% of the applications have at least one timer trigger. As applications can have multiple triggers, the fractions sum to more than 100%. In Figure 3(b), we partition the applications by their combinations of triggers. 43% of the applications have *only* HTTP triggers, and 13% of the apps have *only* timer triggers. Combining the two tables, we find that 15.8% of the applications have timers and at least one other trigger type. For predicting invocations, as we discuss later, while timers are very predictable, 86% of the applications have either no timers or timers combined with other triggers.

3.3 Invocation Patterns

We now look at dynamic function and application invocations. Figure 4 shows the volume of invocations per hour, across the entire platform, relative to the peak hourly load on July 18th. There are clear diurnal and weekly patterns (July 20th, 21st, 27th, and 28th are weekend days), and a constant baseline of roughly 50% of the invocations that does not show variation. Though we did not investigate this specifically, there can be several causes, *e.g.* a combination of human and machine-generated traffic, plain high-volume applications, or the overlapping of callers in different time zones.

Figure 5(a) shows the CDF of the average number of invocations per day, for a representative sample of both functions and applications. The invocations for an application are the sum over all its functions. First, we see that the number of invocations per day varies by over 8 orders of magnitude for functions and applications, making the resources the provider has to dedicate to each application also highly variable.



(a) CDF of daily invocations per function and application, and the corresponding *average* interval between invocations. Shaded regions show applications invoked on average at most once per hour (green, 45% of apps) and at most once per minute (yellow, 81% of apps).

(b) Fraction of total function invocations by the fraction of the most popular functions and applications. Same colors as in Figure 5(a).

Figure 5: Invocations per application and per function for a representative sample of the dataset.

The second observation with strong implications for resource allocation is that the vast majority of applications and functions are invoked, on average, very infrequently. The green- and yellow-shaded areas in the graph show, respectively, that 45% of the applications are invoked once per hour or less on average, and 81% of the applications are invoked once per minute or less on average. This suggests that the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high.

Figure 5(b) shows the other side of the workload skewness, by looking at the cumulative fraction of invocations due to the most popular functions and applications in the sample. The shaded areas correspond to the same applications as in Figure 5(a). The applications in the orange-shaded area are the 18.6% most popular, those invoked on average at least once per minute. They represent 99.6% of all function invocations.

The invocation rates provide information on the average inter-arrival time (IAT) of function and application invocations, but not on the distribution of these IATs. If the next invocation time of a function can be predicted, the platform can avoid cold starts by pre-warming the application right before it is to be invoked, and save resources by shutting it

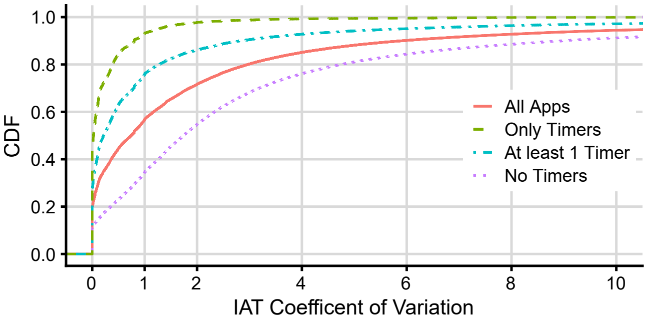


Figure 6: CV of the IATs for subsets of applications.

down right after execution.

Inter-arrival time variability. To gain insight into the IAT distributions of applications, we look at the coefficient of variation (CV) of each application. The CV (standard deviation divided by the mean) provides a measure of the variability in the IATs. We would expect timer-triggered functions to have periodic arrivals, with a CV of 0. Human-generated invocations should approximately follow a Poisson arrival process, with an exponential (memoryless) distribution of IATs [16]. These would ideally yield a CV of 1. CVs greater than 1 suggest significant variability.

Figure 6 shows the distribution of the CV across all applications, as well as for subsets of applications with and without timers. It shows that the real IAT distributions are more complex than the simply periodic or memoryless ones. For example, only ∼50% of the applications with only timer-triggered functions have a CV of 0. Multiple timers with different periods and/or phases will increase the CV. For applications with at least one timer, this fraction is less than 30%, and across all applications the fraction is ∼20%. Interestingly, ∼10% of applications with no timers have CV close to 0, which means they are quite periodic, and should be predictable. This could be due to, for example, external callers (e.g., sensors or IoT devices) that operate periodically. On the other hand, only a small fraction of applications has a CV close to 1, meaning that simple Poisson arrivals are not the norm. These results show that there is a significant fraction of applications that should have fairly predictable IATs, even if they do not have timer triggers. At the same time, these numbers suggest that for many applications predicting IATs is not trivial.

3.4 Function Execution Times

Another aspect of the workload is the function execution time, *i.e.* the time functions take to execute *after they are ready to run*. In other words, these numbers do not include the cold start times. Cold start times depend on the infrastructure to a large extent, and have been characterized in other studies [44].

Figure 7 shows the distribution of average, minimum, and maximum execution times of all function executions on July 15th, 2019. The distributions for other days are similar. The graph also shows a very good log-normal fit (via MLE) to the distribution of the averages, with log mean -0.38 and σ 2.36. We observe that 50% of the functions execute for less than 1s on average, and 50% of the functions have maximum execution time shorter than ∼3s; 90% of the functions take at most 60s, and 96% of functions take less than 60s on average.

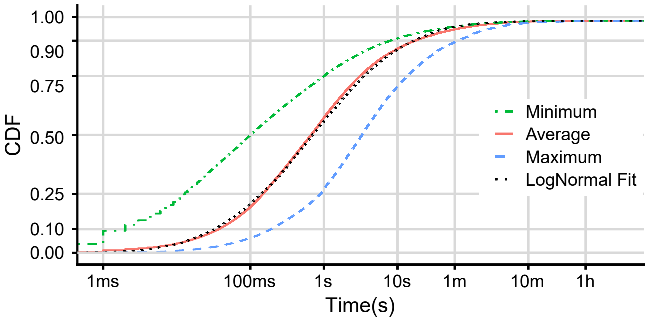


Figure 7: Distribution of function execution times. Min, avg, and max are separate CDFs, and use independent sorting.

The main implication is that the function execution times are at the same order of magnitude as the cold start times reported for major providers [44]. *This makes avoiding and/or optimizing cold starts extremely important for the overall performance of a FaaS offering.*

Another interesting observation is that, overall, functions in this FaaS workload are very short compared to other cloud workloads. For example, data from Azure [12] shows that 63% of all VM allocations last longer than 15 minutes, and only less than 8% of the VMs last less 5 minutes or less. This implies that FaaS imposes much more stringent requirements on the provider to stand-up resources quickly.

Idle times. As we discuss in Section 4, an important aspect of the workload for managing cold starts is idle time (IT), defined as the time between the end of a function's execution and its next invocation. IT relates to IAT and execution time. For most applications, the average execution time is at least 2 orders of magnitude smaller than the average IAT. We verified for the applications in the yellow region in Figure 5(a) – 81% of the applications invoked at most once per minute on average – that indeed the IT and IAT distributions are extremely similar.

Potential correlations. Different triggers had average function execution times differing by about 10×, between 200ms and 2s at the median, but all with the same shape for the distributions. One outlier was a class of orchestration functions with median average execution times of ∼30ms, as they simply dispatch and coordinate other functions.

3.5 Memory Usage

We finally look at the memory demands of applications. Recall that the application is the unit of memory allocation in the platform we study. Figure 8 shows the memory demand distribution, across all applications running on July 15th, 2019. We present three curves drawn from the memory data: 1st percentile, average, and maximum allocated memory for the application. We also plot a reasonably good Burr distribution fit (with parameters $c = 11.652$, $k = 0.221$, and $\lambda = 107.083$) for the average. Allocated memory is the amount of virtual

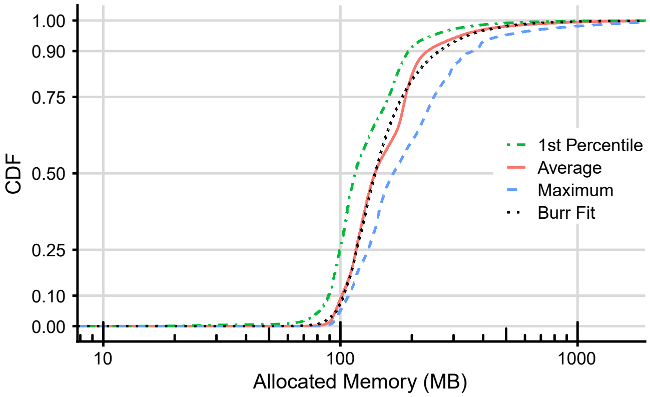


Figure 8: Distribution of allocated memory per application.

memory reserved for the application, and may not necessarily be all resident in physical memory. Here, we use the 1st percentile because there was a problem with the measurement of the minimum, which made that data not usable. Despite the short duration of each function execution, applications tend to remain resident for longer. The distributions for other days in the dataset are very similar.

Looking at the distribution of the maximum allocated memory, 90% of the applications never consume more than 400MB, and 50% of the applications allocate at most 170MB. Overall, there is a 4× variation in the first 90% of applications, meaning that memory is an important factor in warmup, allocation, and keep-alive decisions for FaaS.

Potential correlations. We found no strong correlation between invocation frequency and memory allocation or between memory allocation and function execution times.

3.6 Main Takeaways

From the point of view of cold starts and resource allocation, we now reiterate our three main observations. First, the vast majority of functions execute on the order of a few seconds – 75% of them have a maximum execution time of 10 seconds – so execution times are on the same order as the time it takes to start functions cold. Thus, it is critical to reduce the number of cold starts or make cold starts substantially faster. Eliminating a cold start is the same as making it infinitely fast.

Second, the vast majority of applications are invoked infrequently – 81% of them average at most one invocation per minute. At the same time, less than 20% of the applications are responsible for 99.6% of all invocations. Thus, it is expensive, in terms of memory footprint, to keep the applications that receive infrequent invocations resident at all times.

Third, many applications show wide variability in their IATs – 40% of them have a CV of their IATs higher than 1 – so the task of predicting the next invocation can be challenging, especially for applications that are invoked infrequently.

4 Managing Cold Starts in FaaS

We use insights from our characterization to design an adaptive resource management policy, called *hybrid histogram policy*. The goal is to reduce the number of cold start invocations with minimum resource waste. We refer to a *policy* as a set of rules that govern two parameters *for each application*:

— *Pre-warming window.* The time the policy waits, since the last execution, before it loads the application image expecting the next invocation. A pre-warming window = 0 means that the policy does not unload the application after one of its functions executes. Aggressive pre-warming (a large window) reduces resource usage but may also cause cold starts, in case the next invocation occurs sooner than expected.

— *Keep-alive window.* The time during which an application's image is kept alive after (1) it has been loaded to memory (pre-warming window ≥ 0) or (2) a function execution (pre-warming window = 0). (Note that our definition for this window differs from the keep-alive parameter in fixed keep-alive policies, which is the same for all applications and only starts at the end of function executions.) Longer windows have the potential to reduce cold starts by increasing the chances of an invocation falling into this window. However, this may also waste resources, *i.e.* leave them idle, in case the next invocation does not happen soon after loading.

A *no-unloading policy* would keep every application image loaded in memory all the time (*i.e.*, infinite keep-alive window and pre-warming window = 0). This policy would get no cold starts but would be too expensive to operate.

4.1 Design Challenges

Designing a *practical policy* poses several challenges:

1. **Hard-to-predict invocations.** As Figure 3 shows, many applications are triggered by timers. A timer-aware policy could leverage this information to pre-warm applications right before the next invocation. However, predicting the next invocation is challenging for other triggers.
2. **Heterogeneous applications.** As Figure 5 shows, the invocation frequency and pattern vary substantially across applications. A one-size-fits-all fixed policy is certain to be a poor choice for many applications. A better policy should adapt to each application dynamically.
3. **Applications with infrequent invocations.** Some applications are invoked very infrequently, so an adaptive policy would take some time to learn their invocation patterns. The same applies for applications that it sees for the first time.
4. **Tracking overhead.** Adapting the policy to each application means tracking each application individually. For this reason, the cost to track the information for each application should be small. For example, we need to consider the size of the data structures that will keep this state.
5. **Execution overhead.** Since function executions can be very short (*i.e.*, more than 50% of executions take less than 1 second), running the policy and updating its state need to be fast. This is especially critical considering providers charge users only during their function execution times (*e.g.*, based on CPU, memory). For instance, we cannot take 100 ms to update a policy for each 10 ms-long execution. Due

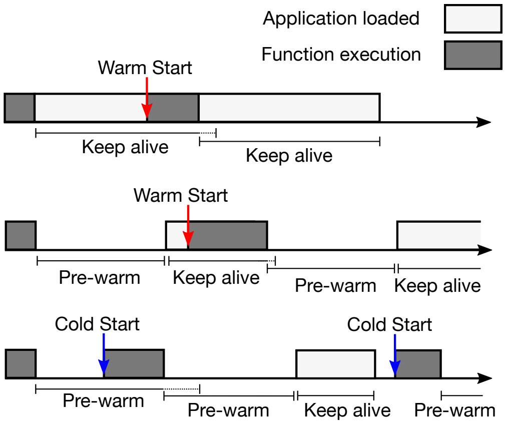


Figure 9: Timelines showing a warm start with keep alives and no pre-warming (top); a warm start following a pre-warm (middle); and two cold starts, before a pre-warm, and after a keep alive (bottom).

to these overheads, expensive prediction techniques, such as time-series analysis, cannot be used for all applications.

4.2 Hybrid Histogram Policy

Overview. Our hybrid histogram policy addresses all the above challenges. To address challenges #1 and #2, our policy adjusts to the invocation frequencies and patterns of each individual application. It identifies the application's invocation pattern, removes/unloads the application right after each function execution ends, reloads/pre-warms the application right before a potential next invocation (after a "pre-warming window" elapses), and keeps it alive for a period (until a "keep-alive window" elapses). The pre-warming window starts after each function execution, and the keep-alive window starts after each pre-warming. If the pre-warming window is 0, we do not unload the application after an execution, and the end of the execution still starts a new keep-alive window. We explain how exactly we compute the length of these windows below.

Figure 9 shows the pre-warming and keep-alive windows in three scenarios. In the top scenario, the pre-warming window is 0, and an invocation that happens before the keep-alive window ends is a warm start. The end of the execution starts a new keep-alive window. In the middle, the next invocation is a warm start, as the application is re-loaded after a pre-warming window. The end of the execution starts a new pre-warming window. In the bottom scenario, there are two cold starts: the first resulting from an invocation arriving before the pre-warming window elapsed, and the second from an invocation arriving after the keep-alive period elapsed.

The policy comprises three main components: (1) a range-limited histogram for capturing each application's "idle" times (ITs); (2) a standard keep-alive approach for when the histogram is not representative, *i.e.* there are too few ITs or the IT behavior is changing (again, note that this differs from a fixed keep-alive policy); and (3) a time-series forecast component for when the histogram does not capture most ITs. Figure 10 overviews our policy and its components. Ultimately, the policy defines the pre-warming and keep-alive windows for each application. Next, we describe each component in turn.

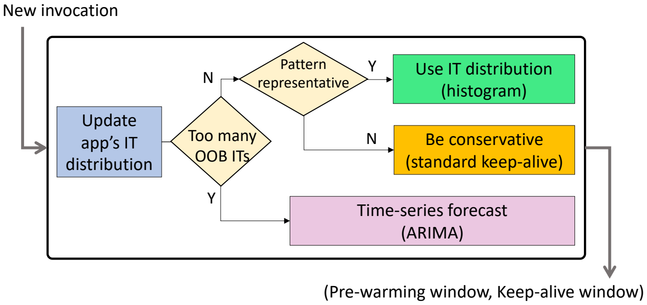

New invocation
Update app's IT distribution
Too many OOB ITs
N
Pattern representative
Y
Use IT distribution (histogram)
N
Be conservative (standard keep-alive)
Y
Time-series forecast (ARIMA)
(Pre-warming window, Keep-alive window)

Figure 10: Overview of the hybrid histogram policy.

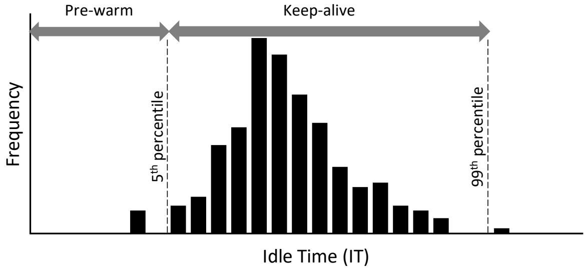


Figure 11: Example application idle time (IT) distribution used to select pre-warming times and keep-alive windows.

Range-limited histogram. To address challenges #4 and #5, the centerpiece of our policy is a compact histogram data structure that tracks the IT distribution for each application. Each entry/bin of the histogram counts the number of ITs of the corresponding length that have occurred. We use 1-minute bins, which strikes a good balance between metadata size and the resolution needed for policy actions. Keep-alive time scales are in orders of minutes for FaaS platforms. We use the same scale for pre-warming. In addition, the histogram tracks ITs of up to a configurable duration (*e.g.*, 4 hours). Any ITs longer than this are considered "out of bounds" (OOBs).

Given the ITs that are within bounds, our policy identifies the head and tail of the IT distribution. We use the head to select the pre-warming window for the application, and the tail to select the keep-alive window. To exclude outliers, we set the head and tail by default to the 5th- and 99th-percentiles of the IT distribution. (When one of these percentiles falls within a bin, we "round" it to the next lower value for the head or the next higher value for the tail.) These two configurable thresholds strike a balance between managing cold starts and resource costs. Figure 11 shows the histogram for a sample application, and the head and tail markers. To give the policy a little room for error, our implementation uses a 10% "margin" by default, *i.e.* it reduces the pre-warming window by 10% and increases the keep-alive window by 10%.

Figure 12 shows nine real IT distributions over a week. The three histograms in the left column show cases where both head and tail cutoffs are easy to identify. These distributions produce the ideal situation: long pre-warm windows and short

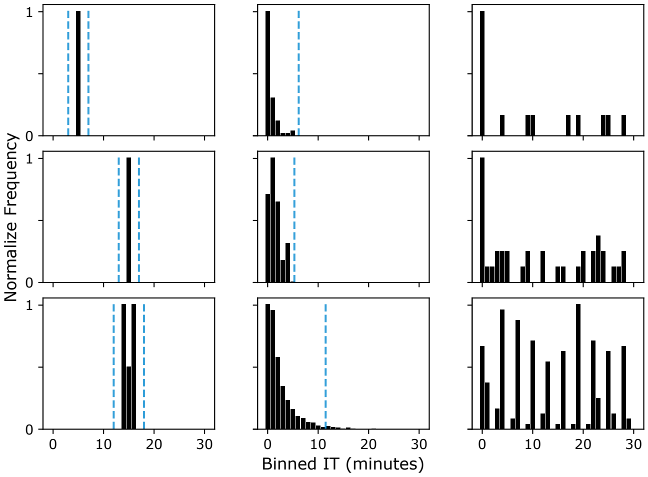


Figure 12: Nine normalized IT distributions from real FaaS workloads over a week.

keep-alive windows. The center cases show no head cutoff as the head marker rounded down to 0. In these cases, the pre-warming window is 0 and the policy does not kill the application after a function execution.

Standard keep-alive when the pattern is uncertain. The histogram might not be representative of an application's behavior when (1) it has not observed enough ITs for the application, or (2) when the application is transitioning to a different IT regime (*e.g.*, change from a consistent pattern to an entirely new one). When the histogram is not representative, we revert to a standard keep-alive approach: pre-warming window = 0 and keep-alive window = range of the histogram (*e.g.*, 4 hours). This conservative choice of keep-alive window seeks to minimize the number of cold starts while the histogram is learning a new pattern. Our policy reverts back to using the histogram when it becomes representative (again).

We decide whether a histogram is representative by computing the CV of its bin counts. A histogram that has a single bin with a high count and all others 0 would have a high CV, whereas a histogram where all bins have the same value would have CV = 0. The histogram is most effective in the former case, where there is a large concentration of ITs (left and center of Figure 12). It is not as effective when ITs are spread widely (right of Figure 12). Thus, if the CV is lower than a threshold, we use the standard keep-alive approach. To track the CV efficiently, we use Welford's online algorithm [45].

Time-series analysis when histogram is not large enough. A compact histogram cannot represent ITs larger than its range. Thus, applications with very infrequent invocations (challenge #3) may exhibit many out-of-bounds ITs. For these applications, our policy uses time-series analysis to predict the next IT. Specifically, we use ARIMA modeling [11].

With an IT prediction, our policy sets the pre-warm window to elapse just before the next invocation and a short keep-alive window. In more detail, we used the `auto_arima` implementation from the `pmdarima` package [2], which automatically searches for the ARIMA parameters (p,d,q) that produce the best fit. As applications using ARIMA are invoked very infrequently, we update the model for each of them after every invocation. To give the prediction some room more inaccuracy, we include a (configurable) margin of 15%. For example, if the predicted IT is 5 hours, we set the pre-warming window to 4.25 hours (5 hours minus 15%) and the keep-alive window to 1.5 hours (15% of 5 hours in each side of the IT prediction).

Justification. Like other FaaS cold start policies, our policy eagerly frees up memory when it is not needed. An alternative would have been to leverage standard (lazy) caching policies, which free up cache space only on-demand. Section 7 explains the differences between these types of policies that justify our approach. Our policy uses a standard keep-alive with a long window, when it does not have accurate IT data about the application, to conservatively prevent cold starts. A shorter window would lower cost but would incur more cold starts. We prefer our approach because it often quickly reduces memory usage greatly, after the histogram becomes active for the application. Instead of using a histogram, we could attempt to predict the next invocation or idle time using time-series analysis or other prediction models. We experimented with some models, including ARIMA, but found them to be inaccurate or excessively expensive for the bulk of invocations. The histogram is accurate, compact, and fast to update. So, we rely on ARIMA only for the applications that cannot be represented with a compact histogram. Producing an ARIMA model is expensive, but can be off the critical path. Moreover, these applications involve only a small percentage of invocations, so computation needs are kept small. Nevertheless, we can easily replace ARIMA with another model.

4.3 Implementation in Apache OpenWhisk

We implement our policy in Apache OpenWhisk [34], which is the open-source FaaS platform that powers IBM's Cloud Functions [21]. It is written in Scala.

OpenWhisk architecture. Figure 13 shows the architecture of OpenWhisk [35]. It exposes a REST interface (implemented using Nginx) for users to interact with the FaaS platform. A user can create new functions (*actions* in OpenWhisk terminology), submit new invocations (*activations* in OpenWhisk terminology), or query their status. Here, we focus on function invocation and container management. Invocation requests are forwarded to the Controller component, who decides which Invoker should execute each function instance. This logic is implemented in the Load Balancer, which considers the health and available capacity of the Invokers, as well as the history of prior executions. The Controller sends the function invocation request to the selected Invoker via the distributed messaging component (implemented using Kafka). The Invoker receives the invocation request, starts the function in a Docker container, and manages its runtime (including when to stop the container). By default, each Invoker implements a fixed 10-minute keep-alive policy, and informs the

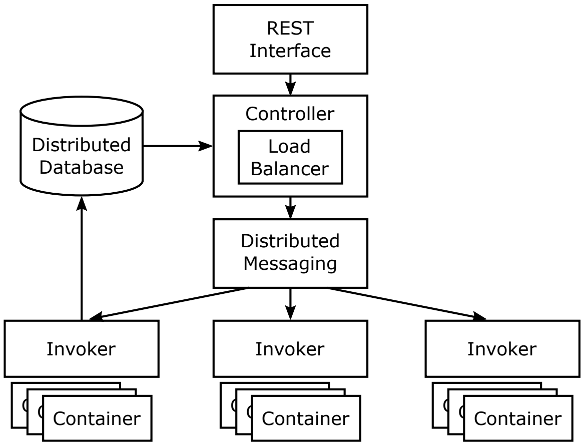


Figure 13: OpenWhisk architecture.

Controller when it unloads a container.

Implementing our policy. We modify the following OpenWhisk components to implement the hybrid policy:

1. **Controller:** Since all invocations pass through the Load Balancer, it is the ideal place to manage histograms and other metadata required for the hybrid policy. We add new logic to the Load Balancer to implement the hybrid policy and to update the keep-alive and pre-warm parameters after each invocation. We also modify the Load Balancer to publish the pre-warming messages.
2. **API:** We send the latest keep-alive parameter for a function to the corresponding Invoker alongside the invocation request. To do this, we add a field to the *ActivationMessage* API, specifying the keep-alive duration in minutes.
3. **Invoker:** The Invoker unloads Docker containers that have timed-out in the *ContainerProxy* module. We modify this module to unload containers based on the keep-alive parameter received from *ActivationMessage*.

5 Evaluation

5.1 Methodology

Simulator. Evaluating our policy requires (1) long executions to assess applications with infrequent invocations, and (2) exploring a large space of configurations. To limit the evaluation time, we use simulations. We build a simulator that allows us to compare various policies using real invocation traces.

The simulator generates an array of invocation times for each unique application. It then infers whether each invocation would be a cold start. By default, the first invocation is always assumed to be a cold start. The simulator keeps track of when each application image is loaded and aggregates the wasted memory time for the application, *i.e.* the time when the application's image was kept in memory without actually executing any functions. We conservatively simulate function execution times equal to 0 to quantify the worst-case wasted resource time. We do not have memory usage data for all applications, so we also simulate that applications use the same amount and focus on the wasted memory time.

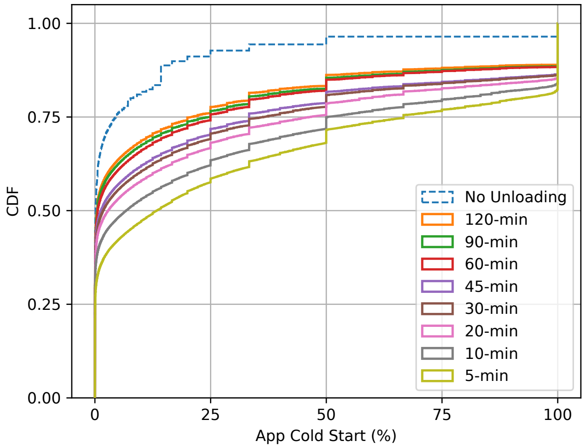


Figure 14: Cold start behavior of the fixed keep-alive policy, as a function of the keep-alive length.

Real experiments. To show that our policy can be easily implemented in real systems with minimal overheads, we use our OpenWhisk implementation (Section 4.3). Our setup consists of 19 VMs. One VM with 8 cores and 8GB of memory hosts containers for the Controller and other main components, including Nginx and Kafka. Each of the remaining 18 VMs has 2 cores and 4GB of memory, hosting an Invoker to actually run the functions in Docker containers.

Workloads. As input to our simulations, we use the first week of the trace from Section 3. For the real experiments, we use a scaled-down version of the trace. We randomly select applications with mid-range popularity. As we run the full system, we limit each OpenWhisk execution to only 8 hours. As we show in Section 5.3, *the experimental and simulation results show the same trends in both cold start and memory consumption behaviors.*

5.2 Simulation Results

Understanding the fixed keep-alive policy. We start evaluating the policy used by most providers: the fixed keep-alive policy. We first assess how the length of the keep-alive affects the cold starts. Figure 14 shows the distribution of cold start percentage experienced by all applications for various lengths. For comparison, we also include a *No unloading* policy, which corresponds to each application only experiencing the initial cold start. Even the *No unloading* policy has ∼3.5% of applications with 100% cold starts; these applications have only one invocation in the entire week.

We see significant cold start reduction going from a 10-minute keep-alive to 1-hour. The 75th-percentile application experiences cold starts 50.3% of the time for the 10-minute keep-alive. This number goes down to 25% for 1-hour. The cold start improvement is more pronounced in the last quartile of the distribution, since applications with infrequent invocations are those that benefit the most. From now on, we will focus on this metric (*i.e.*, 75th-percentile) to report cold starts.

While a longer keep-alive reduces cold starts significantly,

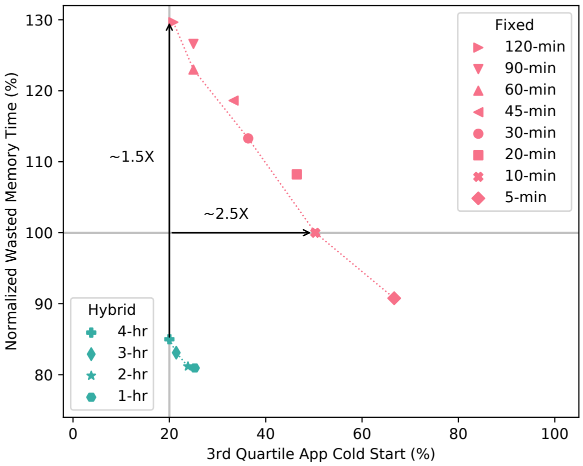


Figure 15: Trade-off between cold starts and wasted memory time for the fixed keep-alive policy and our hybrid policy.

it also increases the resources wasted significantly. The red markers in Figure 15 show the trade-off between cold starts and memory wasted, where we normalize the wasted memory time to the 10-minute keep-alive. The red curve near the red markers approximates the Pareto curve. The figure shows, for example, that a fixed 2-hour keep-alive has almost 30% higher wasted memory time than the 10-minute baseline. An optimal policy would deliver the lowest cold starts with minimum cost. We rely on these Pareto curves to evaluate the policies.

Impact of using a histogram. We now start to evaluate our hybrid policy with the impact of the histogram and its range. The green markers in Figure 15 show the cold start percentage and wasted memory time of our histogram for various ranges. The figure shows how our policy reduces the cold starts significantly with lower memory waste. In fact, the 10-minute fixed keep-alive policy involves ~2.5x more cold starts at the 75th-percentile while using the same amount of memory as our histogram with a range of 4 hours. From a different perspective, the fixed 2-hour keep-alive policy provides roughly the same percentage of cold starts as the 4-hour histogram range, but consumes about 50% more resources. Overall, the hybrid policies form a parallel, more optimal Pareto frontier (green curve) than the fixed policies (red curve).

Impact of the histogram cutoff percentiles. Our policy uses two cutoff percentiles to exclude outliers in the head and tail of the IT distribution. Figure 16 shows the sensitivity study that we used to determine suitable cutoff values. The figure shows that, by setting the head and tail cutoffs to the 5th- and 99th-percentiles of the IT distribution (labeled *Hybrid[5,99]* in the figure), the cold start percentage does not degrade noticeably whereas the wasted memory time goes down by 15%, compared to the case with no cutoff (*Hybrid[0,100]*).

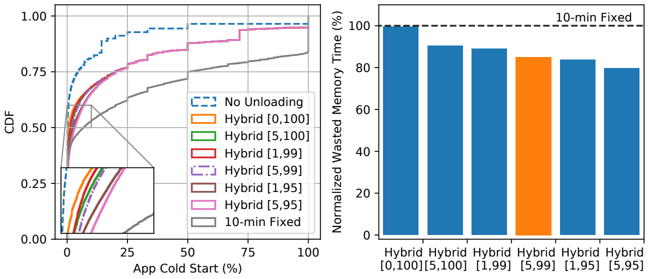


Figure 16: Wasted memory time can be significantly reduced by excluding outliers from the IT distribution.

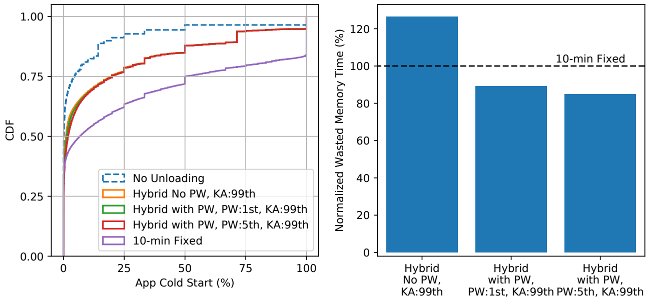


Figure 17: Pre-warming reduces the wasted memory time significantly. The cost is slight increase in cold starts.

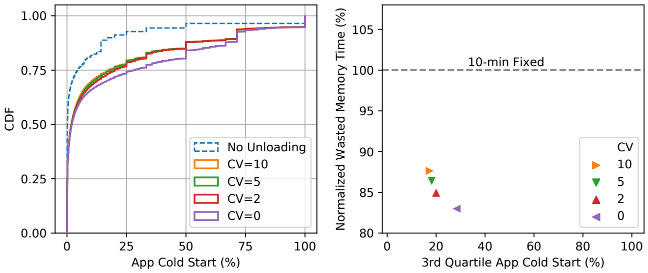


Figure 18: Trade-off between cold starts and memory wasted, as a function of the CV threshold, using a 4-hour range.

Impact of unloading and pre-warming. Complementing our adaptive keep-alive with pre-warming allows unloading of an application right after execution and pre-warming right before the next invocation. This reduces the wasted memory time of application images. Figure 17 shows this, where using similar keep-alive (KA) configurations with and without pre-warming (PW) has significantly different wasted memory time. The cost, however, is adding a small number of cold starts from unexpected invocations. We can control this trade-off by adjusting the histogram head cutoff percentile.

Impact of checking the histogram representativeness. Our policy checks whether the histogram is representative before using it. If the histogram is not representative (*i.e.*, the CV of its bin counts is lower than a threshold), it uses a standard keep-alive approach where applications stay loaded for the same length as the histogram range. We study the impact of different CV thresholds in Figure 18. The figure shows the application cold start distributions (left) and the Pareto frontier (right). We see significant gains using a small CV threshold larger than 0. We opt for CV=2 as our default threshold. Increasing the CV further has negligible cold start reduction with higher resource costs.

Impact of using time-series analysis. Another feature of our hybrid policy is to use ARIMA modeling for applications

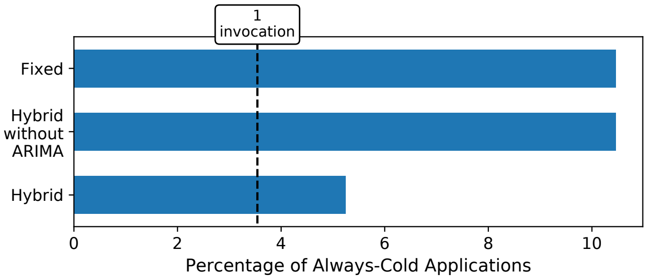


Figure 19: Percentage of applications that always experience cold starts, as a function of policy.

that have many ITs outside the range of the histogram. To evaluate its impact, we now focus on the percentage of applications that show 100% cold starts. Figure 19 shows this percentage when using (1) the fixed keep-alive policy, (2) the hybrid policy without ARIMA, and (3) the full hybrid policy (including ARIMA). All of them use 4 hours for the fixed keep-alive and the histogram range. During the week-long simulation window, 0.64% of invocations were handled by ARIMA, and 9.3% of applications used ARIMA at least once. Using ARIMA reduces the percentage of applications that experience 100% cold starts by about 50%, *i.e.* from 10.5% to 5.2% of all applications. A significant portion of these applications have only one invocation during the entire week and no predictive model can help them. Excluding these applications, the same reduction becomes 75%, *i.e.* from 6.9% to 1.7% of all applications. This shows that ARIMA provides benefits for applications that cannot benefit from a fixed keep-alive or a histogram-based policy.

Summary. Our hybrid policy can reduce the number of cold starts significantly while minimizing the memory cost. We achieve these positive results despite having deliberately designed our policy for simplicity and practicality: (1) histogram bins have a resolution of 1-minute, (2) histograms have a maximum range, (3) they do not require any pre-processing or complicated model updates, and (4) when the histogram does not work well, we resort to simple and effective alternatives.

5.3 Experimental results

We ran two experiments with 68 randomly selected mid-range popularity applications from our workload on our 19-VM OpenWhisk deployment: one experiment with the default 10-minute fixed keep-alive policy of OpenWhisk, and another with our hybrid policy and a 4-hour histogram range. Each experiment ran for 8 hours. During the 8-hour period, there are a total of 12,383 function invocations. We use FaaSProfiler [1, 38] to automate trace replay and result analysis.

Figure 20 compares the cold start behavior of the hybrid and 10-minute fixed keep-alive policies. The significant cold start reductions follow the *same trend as our simulations* (left graph of Figure 16). On average and across the 18 Invoker VMs, the hybrid policy reduced memory consumption of worker containers by 15.6%, which is also *consistent with our simulation results* (right graph of Figure 16). Moreover, the hybrid policy reduced the average and 99-percentile function execution time 32.5% and 82.4%, respectively. This is due to a secondary effect in OpenWhisk, where the language runtime bootstrap time is eliminated for warm containers.

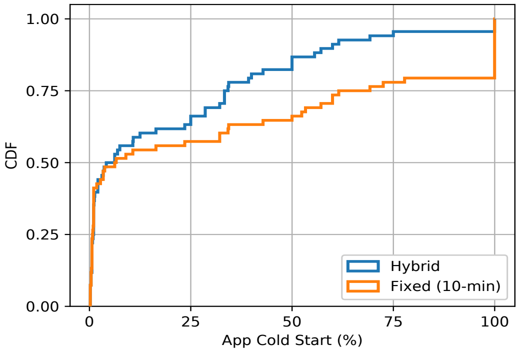


Figure 20: Cold start behavior of fixed keep-alive and hybrid policies in OpenWhisk.

Policy overhead. We measure the (1) additional latency induced by our implementation and (2) the impact of our policy on the scalability of the OpenWhisk controller. The Scala code that implements our policy in the Controller adds an average of only $835.7\mu s$ ($\sigma = 245.5\mu s$) to the end-to-end latency. This overhead is negligible compared to the existing latency of OpenWhisk components: the (in-memory) language runtime initiation takes $O(10ms)$ and the container initiation takes $O(100ms)$ for cold containers [38]. For the uncommon cases where ARIMA is required (0.7% of invocations), the initial forecast involves building the model, which takes an average of 26.9ms, whereas subsequent forecasts take an average of 5.3ms. Since ARIMA works for applications that would normally experience cold starts, these overheads represent a relatively small cost compared to the cold start overhead.

In terms of scalability, CPU utilization is the limiting factor for the Controller. Our policy adds only a 4-6% higher utilization for a range of benchmarking request rates (10rps to 300rps), compared to OpenWhisk's default policy.

6 Production Implementation

We have implemented our policy in Azure Functions for HTTP-triggered applications; its main elements will be rolling out to production in stages starting this month. Here, we overview the implementation.

Azure Functions has a controller that communicates with function-execution workers through HTTP, and a database for persisting system state. The controller gets asynchronous updates from the workers at fixed intervals; we use these to populate the histogram. We keep the histogram in memory (bucket of 240 integers per application, or 960 bytes) and do hourly backups to the database. We start a new histogram per day in the database so we can track changes in application's invocation pattern, and remove histograms older than 2 weeks. We can potentially use these daily histograms in a weighted fashion to give more importance to recent records.

When an application changes state from executing to idle, we use the aggregated histogram to compute its pre-warm interval and schedule an event for that time (minus 90 seconds). Pre-warming loads function dependencies and performs JIT where applicable. Some steps, like JIT of the function code, happen when the actual invocation comes in as the function's code cannot be executed as part of warmup to preserve execution semantics. Each worker maintains the keep-alive duration separately, depending on how long it has been idle for. We make all policy decisions asynchronously, off the critical path to minimize the latency impact on the invocation. This includes updating the in-memory histogram, backing up the histogram to the database, scheduling pre-warming events, and controlling the workers' keep alive intervals.

7 Related Work

There is a fast-increasing number of studies on different aspects of serverless computing. The most relevant for our paper are those that characterize FaaS platforms and applications, and those that propose and optimize FaaS serving systems.

FaaS characterization. A few studies [7, 15, 24–26, 44] have characterized the main commercial FaaS providers, *but only from the perspective of external users.* They typically reverse-engineer aspects of FaaS offerings, by running benchmark functions to collect various externally visible metrics. Our characterization is orthogonal to these works, as we provide a longitudinal characterization of the entire workload of a large cloud provider from the provider's perspective. *Our characterization is the first of its kind.*

Another class of studies looks at the ways developers are using FaaS offerings, by looking at public application repositories [41]. While valuable, this approach cannot offer insights on the aggregate workload seen by a provider.

Optimizing FaaS serving. Another set of relevant work considers optimizing different aspects of FaaS systems. Van Eyk *et al.* [42] identify performance-related challenges, including scheduling policies that minimize cold starts. They also identify the lack of execution traces from real FaaS platforms as a major obstacle to addressing the challenges they identified.

For optimizing each cold start, Mohan *et al.* [32] find that pre-allocating virtual network interfaces that are later bound to new function containers can significantly reduce cold start times. SOCK [33] proposes to optimize the loading of Python functions in OpenLambda by smart caching of sets of libraries, and by using lightweight isolation mechanisms for functions. SAND [3] uses application-level sandboxing to prevent the cold start latency for subsequent function invocations within an application. Azure Functions warms all functions within an application together; thus this was not a concern for us. Replayable Execution [43] proposes checkpointing and sharing of memory among containers to speed up the startup times of a JVM-based FaaS system. Kaffes *et al.* [22] propose a centralized core-granular scheduler. *Our work on reducing the number of cold starts and resource usage by predicting function invocations is orthogonal to these improvements.*

Other studies also use prediction to optimize different aspects. Work in [19, 20] proposes a policy for deciding on function multi-tenancy, based on a predictive model of resource demands of each function. Without discussing design details, EMARS [37] proposes using predictive modeling for allocation of memory to serverless functions. Kesidis [23] proposes to use the prediction of the resource demands of functions to enable the provider to overbook functions on containers. In contrast, we track invocation patterns and use this knowledge to reduce cold starts and memory waste.

Cache management. Finally, one might think that the problem of managing cold starts is similar to managing caches of variable-sized objects, such as Web page caches and others [4, 8, 36]. However, there are two fundamental differences. First, FaaS frameworks are often implemented on top of services that charge by the time resources are allocated (*e.g.*, each application is packaged as a container and deployed to a container service). Thus, cold start policies proactively unload applications/functions from memory, instead of waiting for other applications/functions to need the space. Our policy is closest to a class of TTL-based caches where new accesses reset the TTL [9, 10]. These works did not consider temporal prefetching, the equivalent of our pre-warming. Other caching work did consider it, but with capacity-based replacements [46]. Second, most caching algorithms to date have focused on aggregate performance metrics [13, 14], such as the weighted sum or average of per-object miss ratios. In contrast, we tailor our cold start management to each application to maximize individual customer satisfaction.

8 Conclusion

In this paper, we characterized the entire production FaaS workload of Azure Functions. The characterization unearthed several key observations for cold start and resource management. Based on them, we proposed a practical policy for reducing the number of cold starts at a low resource cost. We evaluated the policy using both simulations and a real implementation, and real workload traces. Our results showed that the policy can achieve the same number of cold starts at much lower resource cost, or keep the same resource cost but reduce the number of cold starts significantly. Finally, we overviewed our policy's implementation in Azure Functions. We released sanitized traces from our characterization data at [31].

Acknowledgements

We would like to thank our shepherd, George Amvrosiadis, and the anonymous reviewers for helping us improve this paper. We also thank Daniel Berger, Bill Bolosky, and Willy Zwaenepoel for their comments on earlier versions of it.

References

[1] FaaSProfiler. http://parallel.princeton.edu/FaaSProfiler.html.

[2] Pmdarima. https://github.com/alkaline-ml/pmdarima.

[3] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. USENIX ATC, 2018.

[4] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A Survey of Web Caching and Prefetching. *International Journal of Advances in Soft Computing and its Applications*, 3(1), 2011.

[5] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[6] Amazon. Invoking AWS Lambda Functions. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html.

[7] Timon Back and Vasilios Andrikopoulos. Using a Microbenchmark to Compare Function as a Service Solutions. ESOCC, 2018.

[8] Abdullah Balamash and Marwan Krunz. An Overview of Web Caching Replacement Algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004.

[9] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman. Adaptive TTL-Based Caching for Content Delivery. *IEEE/ACM Transactions on Networking*, 26(3):1063–1077, 2018.

[10] Daniel Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact Analysis of TTL Cache Networks. *Performance Evaluation*, 79:2 – 23, 09 2014.

[11] George EP Box and David A Pierce. Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models. *Journal of the American Statistical Association*, 65(332), 1970.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. SOSP, 2017.

[13] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Sadoc Menasche, and Yong Chiang Tay. A Utility Optimization Approach to Network Cache Design. *IEEE/ACM Transactions on Networking*, 27(3):1013–1027, 2019.

[14] Andrés Ferragut, Ismael Rodríguez, and Fernando Paganini. Optimizing TTL Caches Under Heavy-tailed Demands. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):101–112, 2016.

[15] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. Performance Evaluation of Heterogeneous Cloud Functions. *Concurrency and Computation: Practice and Experience*, 30(23), 2018.

[16] Robert G Gallager. *Stochastic Processes: Theory for Applications*. 2013.

[17] Google. Google Cloud Functions. https://cloud.google.com/functions/.

[18] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with OpenLambda. HotCloud, 2016.

[19] Mohammad Reza Hoseiny Farahabady, Javid Taheri, Zahir Tari, and Albert Y Zomaya. A Dynamic Resource Controller for a Lambda Architecture. ICPP, 2017.

[20] Mohammad Reza Hoseiny Farahabady, Albert Y Zomaya, and Zahir Tari. A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *Transactions on Parallel and Distributed Systems*, 29(7), 2017.

[21] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions.

[22] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized Core-Granular Scheduling for Serverless Functions. SoCC, 2019.

[23] George Kesidis. Temporal Overbooking of Lambda Functions in the Cloud. *arXiv preprint arXiv:1901.09842*, 2019.

[24] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications. SAC, 2020.

[25] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of Production Serverless Computing Environments. CLOUD, 2018.

[26] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. IC2E, 2018.

[27] Garrett McGrath and Paul R Brenner. Serverless Computing: Design, Implementation, and Performance. ICDCSW, 2017.

[28] Microsoft. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[29] Microsoft. Azure Functions Triggers and Bindings Concepts. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings.

[30] Microsoft. What are Durable Functions? https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[31] Microsoft Azure and Microsoft Research. Azure Functions Traces. https://github.com/Azure/AzurePublicDataset.

[32] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. HotCloud, 2019.

[33] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-optimized Containers. USENIX ATC, 2018.

[34] OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[35] Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md.

[36] Stefan Podlipnig and Laszlo Böszörmenyi. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[37] Aakanksha Saha and Sonika Jindal. EMARS: Efficient Management and Allocation of Resources in Serverless. CLOUD, 2018.

[38] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-Service Computing. MICRO, 2019.

[39] Mikhail Shilkov. Cold Starts in AWS Lambda. https://mikhail.io/serverless/coldstarts/aws/.

[40] Mikhail Shilkov. Cold Starts in Azure Functions. https://mikhail.io/serverless/coldstarts/azure/.

[41] Josef Spillner. Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository. *arXiv preprint arXiv:1905.04800*, 2019.

[42] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. ICPE, 2018.

[43] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. EuroSys, 2019.

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. USENIX ATC, 2018.

[45] BP Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3), 1962.

[46] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal Prefetching Without the Off-Chip Metadata. MICRO, 2019.

Lessons Learned from the Chameleon Testbed

Kate Keahey[1] Jason Anderson[2] Zhuo Zhen[2] Pierre Riteau[3] Paul Ruth[4]
Dan Stanzione[5] Mert Cevik[4] Jacob Colleran[2] Haryadi S. Gunawi[2] Cody Hammock[5]
Joe Mambretti[6] Alexander Barnes[5] François Halbach[5] Alex Rocha[5] Joe Stubbs[5]
[1]*Argonne National Laboratory* [2]*University of Chicago* [3]*StackHPC Ltd*
[4]*RENCI UNC Chapel Hill* [5]*Texas Advanced Computing Center* [6]*Northwestern University*

Abstract

The Chameleon testbed is a case study in adapting the cloud paradigm for computer science research. In this paper, we explain how this adaptation was achieved, evaluate it from the perspective of supporting the most experiments for the most users, and make a case that utilizing mainstream technology in research testbeds can increase efficiency without compromising on functionality. We also highlight the opportunity inherent in the shared digital artifacts generated by testbeds and give an overview of the efforts we've made to develop it to foster reproducibility.

1 Introduction

The primary goal of computer science (CS) experimental testbeds is to support CS systems research by inventing and operating a scientific instrument on which such research can be conducted. Like in any other experimental science, such instrument is a critical tool: while we can conceive of all sorts of experiments, in practice we can carry out only those that are supported by an instrument that allows us to deploy, capture, and record relevant phenomena. The objective of experiment support can be considered along two dimensions: supporting the broadest possible set of experiments for the largest possible set of experimenters. The factors that influence the former include providing state-of-the-art hardware at appropriate scales and sufficiently expressive interfaces for allocating and configuring that hardware (i.e., deploying experiments). The latter is influenced by the cost of per user and per experiment support, but also the usability and familiarity of interfaces that lower the entry barrier for most users.

In this paper, we describe Chameleon, a testbed for CS research and education, and evaluate it from the perspective of the two dimensions outlined above. Chameleon gives users access to a broad array of state-of-the-art hardware, supports deep reconfigurability and experimentation at scale as well as isolation, preventing one experiment from impacting another. Since its public availability date in July 2015, Chameleon has supported 4,000+ users working on 600+ projects.

Unlike traditional CS experimental systems such as Grid'5000 [1], Emulab/CloudLab [2, 3], or GENI [4], which have generally been configured by technologies developed in-house, Chameleon adapted the OpenStack mainstream open-source cloud technology to provide its capabilities. This has a range of practical benefits, such as familiar interfaces for users and operators (or workforce development potential for those not familiar with OpenStack, as they acquire transferable skills), the opportunity to leverage the contributions from a large development community, and the potential to contribute back to that community in turn and thus influence infrastructure used by many users worldwide. Beyond practical benefits, configuring an experimental platform as a cloud also provides a direct answer in the debate over whether CS systems research can be supported on clouds, including potentially commercial clouds. More importantly, it also provides a means of influencing that debate through direct mainstream contributions, i.e., describing how a cloud needs to be configured to support this type of research. A secondary contribution of our paper is thus an articulation of a mainstream cloud configuration that yields a platform suitable for systems research.

Perhaps the most important lesson learned from Chameleon was that testbeds generate a wealth of experimental digital artifacts compatible with the testbed – such as images, orchestration templates, or more recently, computational notebooks – that can be used to re-play experiments. Testbeds are thus "readers" for digital representations of experiments. This creates an opportunity for developing a sharing ecosystem in which users can easily share and replicate each other's experiments and thereby another dimension in which a testbed can support research. This dimension is influenced by how easily experiments can be expressed in a shareable and replicable form – and then how easily they can be discovered and published. We introduced these mechanisms in Chameleon over the last year; while they have been around for too short a time to provide a comprehensive evaluation, we will discuss both their structure and potential.

2 Chameleon in a Nutshell

The Chameleon testbed consists of two operating sites: one at University of Chicago (UC) and the other at Texas Advanced Computing Center (TACC). Our approach to hardware seeks to balance scale (i.e., support for HPC and Big Data experiments) and diversity. Scale was achieved via investment in 12 Haswell racks (2 at UC, 10 at TACC) containing a mix of compute and storage nodes, one with IB interconnect. More recently, they were augmented by 3 SkyLake racks (2 at UC, 1 at TACC) and 1 CascadeLake rack at TACC. The SkyLake racks are equipped with Corsa switches enabling experimentation with Software Defined Networks (SDN). The sites are connected by a 100G network supporting experimentation with large flows. This investment in scale is supplemented by smaller clusters of nodes supporting specialized experiment types: they include four types of GPUs (M40s, K80s, P100s, and P100 with NVLINK), FPGAs, low-power nodes (ARMs, Atoms, and low-power Xeons), and memory hierarchy nodes equipped with almost a TB of RAM memory, and a range of NVMes, SDDs, and HDDs. Over 4 PB in global storage capability is additionally distributed across the sites. About a year ago a Chameleon Associate Site (contributed on a voluntary basis) was added at Northwestern with a modest allocation of nodes equipped with 100G cards, expanding Chameleon's ability to support experiments with large networking flows. A fine-grained and rigorously up-to-date description of hardware can be obtained from the Chameleon Resource Discovery services [5].

Chameleon's capabilities are designed to allow experimenters to allocate and configure these resources at multiple entry levels: users can make allocations expressed as model-based constraints, allocate by node type, or point to a specific node, either on-demand or via advance reservations. Allocatable resources range over nodes, networks, and IP addresses. Resource configuration is supported at bare metal level and supports custom kernel boot. The latter is achieved via the support of whole-disk images, which include a kernel, a partition map, and a bootloader. Reimaging takes places as a sequence of booting to a provisioning ramdisk via PXE, image transfer to node's disk via iSCSI, and a local boot. This allows kernel developers to replace the kernel as needed for experimentation; they can also use serial console access at boot time for debugging and snapshotting to save the revised image. In addition, Chameleon supports network stitching and SDN experimentation via the recently introduced Bring Your Own Controller (BYOC) [6] abstraction built on top of the Corsa DP2000 series OpenFlow switches. Users can create SDN networks that are isolated from each other and the testbed management networks by dynamically provisioned virtual forwarding contexts (VFCs), each with its own controller and subset of ports/VLANs, that are then connected to the nodes and external circuits reserved by the user.

Configuration can also be carried out at multiple entry levels: users can reimage individual nodes and then configure their experiments manually, use orchestration capabilities to render complex (potentially distributed) experiments that can be automatically and repeatedly deployed, or use Jupyter notebooks in combination with one or both of these mechanisms. Monitoring is supported partly by OpenStack Gnocci service that has been augmented to provide capabilities such as e.g., fine-grained power monitoring. The CHameleon Infrastructure (CHI), which implements these capabilities, is built primarily on top of the mainstream open-source OpenStack platform [7] (with extensions and contributions from our team), but also integrates resource representation, versioning and management tools from the Grid'5000 project [1], and network management tools from the ExoGENI [8] project. The implementation of CHI has been described in detail in [9].

Chameleon projects are given allocations of 20,000 SUs (one SU is a node hour) for six months; this aims to strike a balance between what might represent a reasonable amount of time needed to obtain a result (a very variable measure!) and 1% of 6 months' capacity of the initial testbed deployment (i.e., if everybody was using their allocation at the same time the testbed could support no more than 100 projects in that amount of time). Allocations can be recharged (more SUs) or renewed (longer time). In addition, user's leases on the system (i.e., the length of time for which resources can be allocated) are limited to at most 7 days; these can also be extended either programmatically or via interactions with Chameleon operators.

Figure 1 provides a rough summary of Chameleon's growth in hardware and utilization, as well as users and projects. Over the almost four and a half years of its operational life Chameleon has supported 4,331 users across 655 unique projects representing a broad array of research and educational uses. The growth of Chameleon usage has been steady since the project start, with about 7 new projects per month in the first years of operations, accelerating to eleven new projects per month in 2019. Incremental hardware investments have been keeping up with that growth. We also observe a similar trend to that already reported by [3] where usage is lower during summer and winter breaks and peaks during the semester and important conferences such as the supercomputing conference series.

3 Most Experiments for Most Experimenters

We now evaluate the key decisions made in Chameleon from the perspective of supporting the most experiments for the most experimenters. To evaluate the former, we will look at whether the hardware and capabilities we provide are sufficient to support experiments in our community; for the latter we will look at quantitative measures such as the numbers of users and experiments the tested can simultaneously support and the types of projects it attracted.

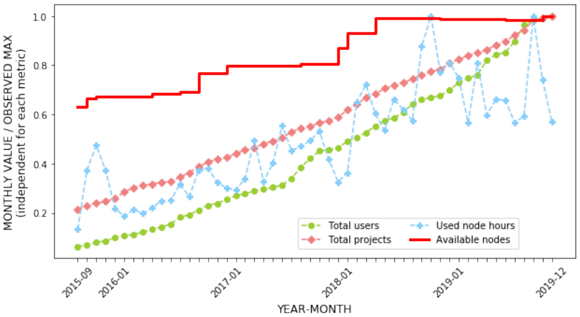


Figure 1: Increasing usage and capacity of the testbed over time. Monthly values are normalized to the maximum observed value over time. Scalings are independent for each metric.

3.1 Experiments

In this section we describe and evaluate both our strategy for hardware configuration and the technical decisions we made to provide access to this hardware such that it supports the broadest possible set of experiments. Unless we specify otherwise, the data reflects period between 09/01/15 to 12/31/19.

3.1.1 Hardware

Where resources are limited (as they inevitably are in academic testbeds) it is important that hardware investments strike a balance between scale, i.e., support for large-scale experimentation, such as HPC, and diversity, i.e., support for a broad range of resources on which different research questions can be tried. Given these considerations, our hardware investment strategy was to build up scale at the beginning of the project (the original 12 rack Haswell deployment) and then introduce diversity gradually with an eye towards the types of projects that were requested the most, and the type of resources that were utilized the most. We also held back a "strategic reserve" of hardware investment to develop the testbed as innovative hardware solutions emerged.

To understand how well we satisfy the need for scale we looked at the size of lease requests for the testbed in general, and the large Haswell partition in particular. By far the most leases on the testbed are requests for a single node, constituting 67.19% of testbed leases (63.24% of Haswell leases) with only 5% of requests exceeding 10 nodes (11% for Haswell). The largest lease on the testbed was 120 nodes. Overall, this means that our investment in scale did indeed broaden the set of supported experiments while also allowing us to support more simultaneous leases.

To understand how well we support the need for diverse hardware types, we show in Figure 2 node availability against the percentage of time that it was available, first for the overall time since resource installation, and then during the busiest and least busy month as measured by highest and least utilization of the testbed as a whole (Cascade Lake was installed very recently and is not shown). By far the most used resource types are the four GPU types, with the two newer GPUs (GPU P100) more in demand than the two older ones (GPU K80 and M40), followed closely by the memory hierarchy nodes. On the other end of the spectrum, ARMs, Atoms, and low-power Xeons—targeting specialized power experiments—are used the least. The large-scale resources, Haswells and Skylakes, as well as the FPGAs, occupy the middle.

While these categorizations are roughly consistent across specific time periods and overall, some important details are different. For example, the availability of the GPU P100 NVLINK is less in both most and least used month than the overall availability graph would suggest; this is due in significant part to the fact that resources are typically much less utilized immediately after they are introduced as the knowledge of their availability has yet to be absorbed. In general, periodic usage patterns might fluctuate due to specific conference deadlines or teaching/workshop use.

One lesson learned from our experiences is the need for adaptation. Overall, introducing diversity to the testbed gradually allowed us to make changes to respond to community demand, shaped by the evolving nature of the research needs. For example, based on the response to our early K80 and M40 deployments driven by emergent interest in machine learning, and combined with less demand for a low-power processors, we re-budgeted some of our planned investment between those categories and also invested our "strategic reserve" into more GPUs. This meant that we were able to provide ample support for low-power experimentation while keeping up with emergent demands. Because of periodic fluctuations it is hard to formulate recommendations on when such change should be considered; based on our experiences it is likely to occur when usage moves to the area of the graph between the large-scale and memory hierarchy resources.

3.1.2 Capabilities

Resource Description. Much of the usefulness of the testbed relies on the manner in which users gain access to hardware. To do that, users have to describe the resources they need. Commercial clouds offer a variety of "instances", sometimes with vaguely described properties (e.g., "high I/O bandwidth"). Experimental testbeds allow users to choose a specific hardware type and sometimes seek to ensure that all servers of the same type have comparable performance [3,10]. In contrast, we take the view that performance variability is a fact of life and often a research topic in itself. Our approach therefore is to allow users to choose resources on a range of levels: from a model description expressed as a set of constraints (e.g., "memory of at least X" or "X nodes situated on the same rack"), through describing hardware type (e.g., a Skylake node), or by referring to a specific node.

Analyzing Chameleon lease requests made between 09/16

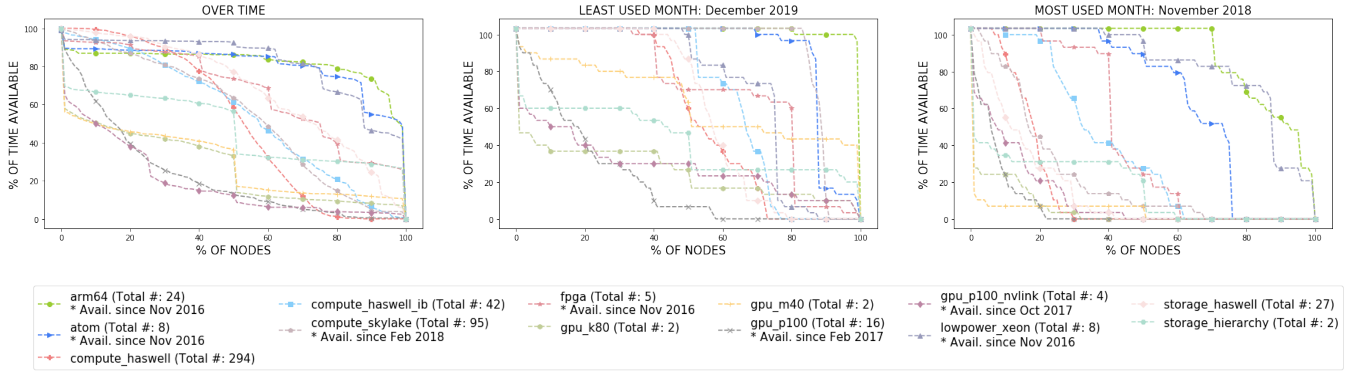


Figure 2: The usage of Chameleon hardware types expressed as resource availability plotted against the percentage of time in which it was available for (a) overall availability since resource installation, (b) the least and (c) most utilized month.

and 11/19, we find that the majority (89.24%) were created using a single constraint (the rest are leases using either no constraint or multiple constraints; 9.5% and 1.26%, respectively). Of the single constraint leases, 90.18% were created by specifying the hardware type and only 3.38% specify node uid (a specific node). However, when we look at single-constraint leases that are created more than 7 days in advance, the percent of leases with node type constraint drops to 59.91%, while the percent of leases with uid increases to 18.45%, which shows that users who need a specific node are willing to wait for it rather than replace it with something else.

Overall, a hardware type is clearly the most requested quality. Specific nodes are requested relatively infrequently, though some experimenters do need them and model-based descriptions based on high-level constraints are rare. While it is often tempting to think that a model-based description is the ideal, the following anecdote illustrates the limitations of this approach. Using a model-based request ("memory greater than X"), one of our users was assigned an ARM node; this led to a difficulty since although this was a correct match for the experimental model, the user's tooling did not work on ARMs. In subsequent conversation with our support staff the user was advised to browse our discovery services, found the Chameleon memory hierarchy nodes, and concluded that with those nodes he would be able to design a more ambitious experiment. We derive two lessons from this: first, models are not all that is needed for determining the right experimental resources (logistical concerns need to be taken into account as well); second, resource discovery phase is an essential part of an the experimental workflow and critical to taking full advantage of the testbed.

Allocatable Resources. Another matter of interaction with the testbed consists of being able to obtain resources in a timely manner. Commercial clouds use the metaphor of an "endless resource" always available on-demand – in practice no resource is of course endless even in commercial clouds (e.g., the current initial limit at AWS is 256 VCPUs [11]) – though some are sufficiently large. Thus, the ability to gracefully deal with availability limitations is important, particularly in academic clouds where we try to maximize small scale resource investment.

To provide such ability we introduced the abstraction of an allocatable resource described in [12]. In brief, an allocatable resource allows users to manage a resource allocation in terms of both time and resource assigned to the allocation (i.e., when an allocation starts and ends, and well as how many nodes belong to it). In particular, the ability to manage the start time is sometimes referred to as "advance reservations" and is a generalization of on-demand availability provided by commercial clouds (i.e., on-demand is an advance reservation with start time set to "now"). The resources can be of various types and can be managed to fulfill different conditions; in Chameleon currently the managed allocatable resources consist of nodes, VLANs, and public IP addresses. The implementation of this capability and its contribution to OpenStack was initiated by the Chameleon team.

As [12] demonstrates, allocatable resources, and advance reservations in particular, are useful in providing access to scarce resources: the scarcer the resource, the more likely users are to make an advance reservation to ensure availability, and the longer in advance this reservation is likely to be made. In figure 3, we provide a more detailed demonstration of this concept: we scatter plot all leases made for three types of resources: Haswell compute nodes at TACC (largest partition), Skylake compute nodes at UC (largest partition), and our GPU P100 cluster (16 nodes), noting the number of nodes requested and the reservation lead time.

We see that the GPU P100 nodes (one of the most utilized resources per Figure 3) have by far the longest advance reservation lead times even though only very few users reserve more than one node. On the other hand, Haswell@TACC, used for experiments at scale, show a significantly higher proportion of leases with multiple nodes (with the max being 85). While many of them are created with some lead time, it was

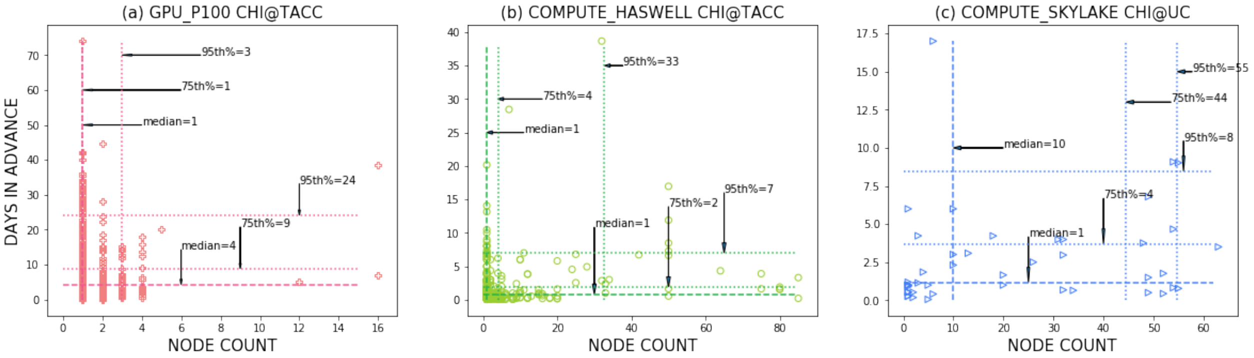


Figure 3: Node counts vs. advance reservation lead time of advanced leases for (a) GPU_P100, (b) Haswell, and (c) Skylake

possible to create some large leases on-demand; they are correlated to summer use (low utilization) and no leases with size in the 95th percentile were available after 2016 as the testbed became popular. This trend is even more pronounced when looking at the Skylake nodes, which are a scarcer resource than Haswells (64 versus 278 for largest partition) but also support experiments at scale. Advance reservations are thus useful in managing two types of resource scarcity: very scarce resources (e.g., GPU P100) will require high lead times, but relatively abundant resources (e.g., Haswells) can become scarce if a large reservation is requested.

The end time of a resource reservation can also be extended programmatically, although according to our policies this can only take place within 48 hours of lease expiration and of course only if the resource is not reserved by another user (a policy exception can also be requested via the help desk.). This last consideration limits the practical usefulness of this feature in the case of scarce resources, as it is likely that they will have been reserved by the time the extension window becomes active. For this reason, programmatic extension requests have only been successful in relatively few cases in practice, e.g., during the last year at UC only 5.4% leases got extended in this way; though half of them more than once.

Separation of allocation and configuration. Unlike commercial clouds where resource allocation and image deployment are one operation, Chameleon separates them to allow users to map different images to an allocation. This capability proved relatively popular with experimenters: 22.07% of the allocations had more than one instance deployed, and roughly half of those (9.17% of all the allocations) had more than one unique instance (i.e., associated with a different image) deployed; the average number of instances deployed within one allocation is 1.45 (with max being 12) and of unique instances is 1.12 (with max being 10).

Networking. Network isolation is an important property in that it allows non-standard IP configuration, potentially disruptive services, or security experiments that analyse or intentionally attack other nodes on the isolated network. Similarly to many of the base Chameleon features, we implemented this via standard OpenStack services; within each geographic site, users can create Chameleon networks that are isolated within unrestricted VLANs (i.e., no firewalls) and logically connect any number and type of nodes.

Chameleon's "Bring Your Own Controller" (BYOC) [6] capability extends these isolated networks by providing direct user control of network flows and configuration via a standard or customized OpenFlow 1.3 controller. Though OpenStack did not support BYOC out of the box, its modular design enabled us to implement the feature as a custom Neutron plugin, which enables users to specify the IP and port of the user's OpenFlow controller (whether provisioned on Chameleon or externally). Chameleon uses Corsa DP2000 series switches to dynamically create isolated OpenFlow 1.3 network slices in hardware operating at full performance (10 Gbps node ports and a 100 Gbps uplink); this is in contrast to an approach which provides coarse control of whole OpenFlow switches from a static pool of hardware as implemented by CloudLab. BYOC enables many experiments from basic hands-on educational experiences with OpenFlow to advanced networking experiments that optimize performance of network traffic or identify and remedy security breaches by analysing low-level traffic behavior. Despite the fact that BYOC networking is new and targeted at highly specialized and advanced users, there have been already been 11 unique projects (representing 4% of active projects over the time period) that have deployed OpenFlow experiments on the Chicago site alone.

Networking experiments on Chameleon are not limited to the Chameleon testbed alone. Users can create dedicated layer 2 circuits between Chameleon networks and external facilities such as ExoGENI, campus laboratories, public clouds, and other Chameleon sites; creating dynamic connections of this type is often called "stitching" [4]. ExoGENI provides a dynamic stitching service that connects a wide collection of participating facilities, including Chameleon. Chameleon users can create isolated stitched links between their networks (including BYOC networks) and ExoGENI and can extend those links across ExoGENI to remote facilities. In addition, multiple stitched links can be connected to a single Chameleon network, enabling user-controlled wide-area multi-path routing experiments. To date, 22 unique projects (representing 8% of active projects over the feature's life time) that rely on network stitching have created 920 stitched links between remote facilities using ExoGENI.

Orchestration. Once a lease is created, users can configure it using Chameleon provided images, create their own (often, but not always, derived from Chameleon provided images), or use complex appliances [13], representing concepts such as a cluster, a cloud, or a networking experiment, that can be deployed "with one click" and then repeated in future deployments, similar to CloudLab profiles [3]. These experiments are configured using images in conjunction with Heat orchestration templates [14] that define how to deploy and contextualize [15] them to create the desired integrated environment and processes. Using Heat, an active Chameleon topology can be modified (by e.g., adding or removing nodes, altering MTUs on the network, or changing a post-boot step for a particular node) through changes to the underlying Heat template; the orchestration system applies the delta without forcing re-creation of the entire topology. The choice to decouple allocation and configuration made this functionality easier: because a set of resources is allocated explicitly to a user for a time period, any topological or software/firmware configuration can vary without breaking the researcher's assumptions about the underlying hardware.

Consistent with our decision to make the testbed available at various levels of access, the use of orchestration is optional for Chameleon users. Using orchestration/Heat provides repeatability at the cost of an additional up-front investment (i.e., developing an orchestration template) and thus tends to be used in later stages of a project when experimental configuration is settled on. Since users often develop orchestration templates by modifying existing ones, the Chameleon project provides 3 complex appliances (images+Heat template) and another 14 individually-supported complex appliances are hosted on our appliance catalog; the most popular are MPI bare-metal cluster (MPICH3), Ryu OpenFlow Controller, and OpenFlow - QuickStart appliance. To date, 81 Chameleon projects have used Heat, among those 20 were in systems, 17 in education, and 12 in networking, with the rest ranging over a variety of topics including security, power management, and others. The usage data since 10/16 when this feature was introduced show a steady upward trend in orchestrated deployments: 94 (2017), 155 (2018), and 405 (2019), though more recently users were increasingly using Jupyter notebooks or scripting for orchestration. Overall, while orchestration is an advanced feature and thus the uptake is slow, it is proving a useful tool to express a range of experiments.

Configuring complex experiments, even when automated via orchestration, can still take significant time, as packages need to be installed, configuration scripts run, and tests executed. Thus, while separating allocation and configuration proved a successful decision, users often ask for functionality that effectively recombines these actions, i.e., automatically triggers the deployment of an orchestrated experiment when a user's advance reservation comes into effect. To provide it, we introduced the automated deployment feature [16] in 01/19 that automatically triggers the deployment of experiments orchestrated with Heat. So far it has been little used while still being often requested; this likely points to the need for more energetic education efforts as well potentially to the need of extending this capability to other orchestration methods, in particular the increasingly popular Jupyter-based approach.

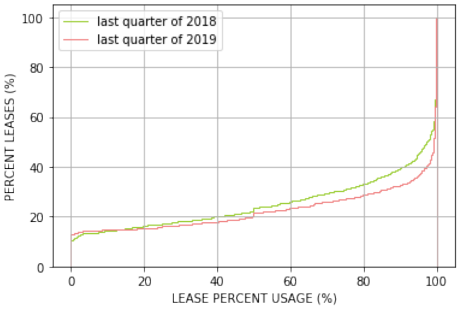


Figure 4: The cumulative distribution functions (CDFs) of lease percent usage for the last quarter of 2018 and 2019.

Managing User Behavior. Perhaps the most significant challenge of operating Chameleon, common to all academic cloud resources, is ensuring fair sharing of the resource. Unlike in HPC datacenters, where actual resource use is tied to the submission of a specific program (such that if the program fails the resource grant is withdrawn), access to cloud resources is given out on an open-ended basis. Commercial clouds create incentive to use no more than is needed by charging for the duration of access. Academic clouds, such as Chameleon, seek to provide a similar incentive via allocation policies (see Section 2); this is generally less successful since users can recharge or renew their allocation relatively easily. In fact, we found this measure to be inadequate on its own early in the project as users created leases to "put a hold" on resources that then went unused, significantly reducing testbed capacity for others. This led us to introduce the (extensible) 7 day lease limit described earlier which improved fair sharing at the cost of imposing extra overhead on experiments that legitimately need more than 7 days. However, we still see leases on the testbed that merely hold resources without using them.

To manage this situation, we introduced a policy whereby users are expected to start using their lease within a certain amount of time from deployment. This policy is enforced by a "lease reaper" (deployed in 09/19) that monitors the use of a lease, sends a reminder to users not using their leases, and terminates them if still unused after a certain period of time.

Figure 4 shows a comparison of lease usage for the last quarter of 2018 and last quarter of 2019 (before and after the lease reaper was introduced). While it is natural that resources in a lease may be unutilized for some time (e.g., between deployment of different images), we see that in 2018 (without lease reaper) a large percentage of leases is underutilizing resources to a significant extent. However, in 2019 (with lease reaper) the situation improved: it went from 40.79% to 45.95% fully-used leases and from 66.92% to 71.19% 80%-used leases. More sophisticated ways of ascertaining if a lease is actually being used are likely to tighten the gap between allocated and used leases further, but at the same time become open to "gaming" by users emulating lease usage via artificial means, reducing their effectiveness. Thus, striking the right balance of incentive and access to promote fair sharing in academic environments is still an open question.

Summary. On the whole, we found that the key design decisions we took, whether by introducing new hardware or new capabilities, led to expanding the set of experiments available to our user community, and were thus quickly embraced. While we still receive new feature requests, they are increasingly smaller and come less often. At the same time, a significant lesson learned in the process of operating Chameleon is that no research testbed is ever complete because the set of desired experiments is constantly expanding. As the research frontier advances emergent research opportunities create the need for new scientific instruments – or new features in existing scientific instruments – to support their exploration. While setting aside a strategic reserve in hardware served us well, the extent to which this phenomenon drove development was surprising: not only did we need to adapt the system to leverage new opportunities in hardware (such as e.g., providing the BYOC capability on top of the Corsa switches), we needed to develop abstractions (such as e.g., the allocatable resources) to integrate those extensions in the experimental workflow. The most significant types of experiments that Chameleon does not support yet are thus in emergent topics – for example, on the intersection of Internet of Things (IoT) and cloud computing as well as machine learning.

3.2 Experimenters

Supporting as many users as possible will be influenced by two factors: how many users a testbed can sustain by managing the cost of users and experiments and how many users it can attract by adapting itself to the needs of different communities; this section will discuss how well Chameleon was able to achieve both.

Isolation and Automation. Providing an appropriate level of isolation captures an important trade-off: it should be fine-grained enough to divide the testbed efficiently between multiple experiments – but also coarse-grained enough to satisfy the isolation needs of a specific experiment. Since the granularity often goes with the level of isolation a specific mechanism provides [12] we must be careful to not sacrifice the required level of isolation; at the same time we want to serve as many users as possible. In Chameleon, we navigate this trade-off by configuring the bulk of the testbed with CHI while setting aside two racks (originally three) provided as a standard OpenStack/KVM cloud. This finer-grained system isolation that this alternative cloud provides means that multiple user VMs can be deployed on one node instead of allocating a whole bare metal node (though it does not provide the performance isolation that a bare metal offers). Because of this efficiency, we were also able to offer a different policy: users can make open-ended deployments on the KVM partition while CHI (bare metal) leases are limited as to 7 days at most. Last but not least, it is more suited to less-experienced users.

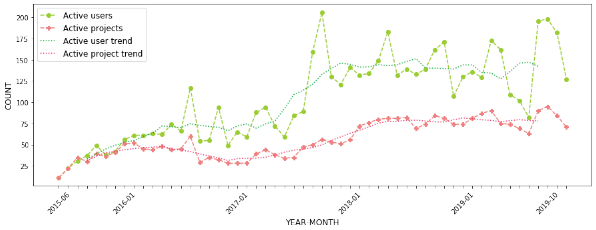


Figure 5: The numbers of active users and projects over time. The trend lines average the number of users/projects over 6 month period.

Our data indicate that 209 Chameleon projects (22.94% of all Chameleon projects) used our KVM partition at least once. Among those projects, 12.92% are projects in CS education. About 26.16% of our total number of users used the partition, most of them assigned to an educational project. Most VMs (71.52%) are deployed for an hour or less and only 3.18% leverage the ability to claim testbed resources for more than one week. The median daily count of deployed VMs is 344 (with max/min of 1490/29); each of those would have likely occupied a bare metal node otherwise. All those statistics point to significant educational use; given the number of users and projects overall this seems a good investment for what currently constitutes only 16% of our total Haswell system and even smaller fraction of the overall testbed.

Supporting Volume. While isolation method determines the unit of sharing, the largest factor in the ability to support as many users as possible is automation since it lowers the per-user and per-experiment cost. We noted earlier that Chameleon is a production testbed, i.e., a testbed that supports production services that yield individual/breakable testbeds. Consistent with this definition, we define Chameleon testbed functions as only those experiments that are accessible to users in an automated manner (while we also support experiments requiring manual intervention from operators and special requests, we consider them support functions, not testbed functions). We now examine indicators of how much user volume the testbed can support.

We first asked how many active users the testbed supported

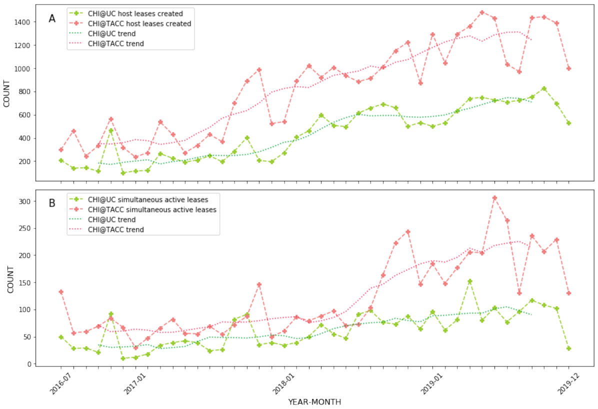


Figure 6: (A) Total leases created each month. (B) Maximum simultaneous active leases by month. The trend lines average the number of leases over 6 month period.

overtime. We see that numbers of active users follow the general pattern of slow and busy months (semesters versus breaks) that we have already seen in Figure 5. However, although we saw that the cumulative number of users was rising, it is interesting to note that the average number of active users and projects grew significantly about two years after the testbed has became available (fall of 2017). It is hard to pinpoint this to any one reason but possibilities include a lag that it takes for a new testbed to become established, the incremental introduction of features that broadened the set of supported experiments, and our first Chameleon User Meeting held just before the trend increase. At peaks, the testbed supported about 200 active users; this is twice as much as the lower bound that our allocation policy is based on (Section 2); luckily not all users are living up to their allocations!

We then asked how many leases and simultaneous leases users were able to create on a per month basis. The result for unique leases is shown in Figure 6 (A). We see that the trends are consistent with the number of active users reflecting the usage patterns in a similar way and picking up around the same time, though the growth trend is still continuing. The result for simultaneous leases (i.e., for each month, we found the max number of leases happening at the same time) is shown in Figure 6 (B); we see that the testbed has sustained up to 300 simultaneous ongoing experiments, but the trends picked up about a year later than trends for active users and experiment counts; it is clear that the testbed is now becoming more saturated.

Support Cost. Another metric important in the discussion of any testbed is support costs, specifically the time effort required by the team. Chameleon users have submitted 3,167 technical help desk tickets, averaging roughly 13 tickets every week and less than 1 ticket per user. On average users receive a reply within 15 hours and their issue is completely resolved within 2 weeks. These trended down over time: in 2019 the average response time was 16 hours, while the resolution time was 6 days. The costliest tickets concern hardware failures, which are often difficult to diagnose and/or require ordering new parts and performing maintenance. Cutting-edge or nonstandard hardware and firmware also pose problems for support staff, as documentation might not be extensive (or even exist) and having expertise at hand is unlikely. In all cases, an ounce of prevention is worth a pound of cure: we have deployed or implemented an operational model integrating early detection (e.g., daily/hourly "happy path" tests of common user flows, live-monitoring with Prometheus [17], and a catalogue of alertable issues and resolution steps) and automated remediation (in the form of "hammers", i.e., bots that periodically check for and fix irregularities in testbed usage and performance). We additionally automate most common operator tasks, such as building new base images, deploying patches or configuration changes to the control plane, and taking nodes in or out of maintenance. Details of Chameleon operations has been published in [18].

Another indicator of cost in a system where users are given significant privileges is the cost of security management. We employ a range of standard security practices designed to make the system more secure: project PIs are vetted and assume responsibility for users on their projects; we provide base images configured and maintained by our team with reasonable security defaults and SSH key pairs for authentication; the management network is isolated from tenant networks; and we use intrusion detection system (IDS) alerts across the entire deployment. Since the system went public we've had a number of security incidents; most are caused by users either unknowingly or deliberately shirking best practices, e.g., using images with a known administrative password or using outdated software with known exploits – the latter sometimes to be compatible with benchmarks and other research software. The most typical types of exploits result in activities such as distributed denial of service (DDoS) attacks (~40% incidents) or bitcoin mining (~36% incidents). Those are typically identified via IDS systems operated on sites and trigger well-defined security procedures in response usually involving user/PI cooperation. So far, we've only had one incident involving malicious users in the first quarter of 2020 which shows that our PI vetting methods work well on the whole. Considering the nature of attacks to date the greatest improvement would probably be effected by more user training in techniques such as setting up bastion host for when insecure researchware has to be used as well as general training in operational security.

Community. Chameleon users come from 168 different institutions, the vast majority of which are US colleges and universities from 40 states and Puerto Rico, including 11 minority serving institutions. While most of the research projects we support are in computer science, 54 identify themselves as being from outside of computing disciplines, primarily in life sciences, astronomy and other fields. By analyzing project abstracts we see that roughly 12% of all projects relate to cybersecurity, 20% are involved in machine learning, 10% are in edge computing or IoT applications, 5% are doing research work relating to containers (scheduling, virtualization, or performance), 2% are investigating software-defined networking. Several hundred grants are reported by users as the source of funding; the vast majority of these are from NSF, and within those the vast majority are spread among all divisions within CISE. About 5% of grants are supported by the DOE, DARPA, or the Air Force Office of Sponsored Research. A handful of projects are supported by industry, and several more have international support.

Publications are a complicated metric to track as they tend to be a lagging indicator; many happen after the allocation is complete, when there are few incentives for users to report them to the project. Further, the primary way we capture publication counts is through self-reporting when users seek a renewal of their allocation; while this is an incomplete method it still represents a useful lower bound. Through 2019, users have self-reported about 275 publications relating to their work on Chameleon, with many projects still active. There are 75 referred journal articles among these, with close to 200 conference publications. The spread of publications is fairly wide, with no clear concentrations in particular conferences or journals. As would be expected for a testbed, growth over time is dramatic and lagging: of the 75 journal papers, only 3 were published in the first project year, 8 in the second, 10 in the third, 23 in the fourth, and 31 have been published so far in the fifth year with several months remaining.

In addition to research usage, there is substantial educational use of the system. 45 projects support classroom instruction, often multiple classes over multiple semesters; all but four of these projects support courses in CS departments at 41 different schools. In total, the education projects have used about 9% of the total time available on Chameleon to date (roughly 675,000 node-hours), and they represent about 9% of the total projects – so an average classroom project uses about 15,000 node-hours, matching the usage of a typical research project.

One measure of the satisfaction of the user community is persistence. Projects are initially allocated time on the system for 6 months, and at the end of the year they must seek a renewal or extension. Of the projects that have reached the end of their initial year of allocation, about three-quarters of all projects have sought to renew their allocations, indicating that they find value in the use of the system. Many projects seek multiple renewals – in fact thirty-three of the original projects from the first year have been renewed multiple times and counting!

4 Building a Testbed on Top of Mainstream Cloud Implementation

When the Chameleon project started, we were presented with the unique opportunity of building the testbed on top of the then maturing cloud infrastructure: the first version of the OpenStack Ironic component [19], implementing bare metal reconfiguration (which we knew would be an indispensable capability of the system) has been released a few months prior to the project start, and while not yet an official part of the system has already been used in some bare metal deployments. The chance to base a testbed for cloud computing research on a mainstream open source implementation held out many possibilities, but will it be enough to support all the experiments that needed to be supported in the way they needed to be supported? After thorough evaluation of the system and development of a few alternative risk-mitigating strategies we decided that capabilities we needed were there—or were within reach in that they could be developed by our team. This section presents an analysis of the advantages and cost of taking this approach.

One practical benefit of using OpenStack is that it provides familiar interfaces to users and operators. The 2018 OpenStack User Survey [20] (most recent) included 858 OpenStack deployments across 441 organizations and 63 countries; of those organizations, 13% were categorized as academic or research-oriented. Those include major scientific institutions such as CERN [21], NeCTAR [22], and NASA JPL [23], and a formal Scientific Special Interest Group (SIG) [24] for OpenStack's use in science domains has existed since 2016 [25]. Since the introduction of the OpenStack Administrator certification three years ago, 3,000 individuals in 77 countries have taken the test [26]. All this not only creates a base of familiarity with OpenStack for users and operators – but also ensures that such familiarity is a transferable skill and thus valuable for workforce development. Finally, the Net Promoter Score [27] of 41 reported by the survey (up from 25 in 2017) indicates that the OpenStack environment continues to improve in terms of usability and that users enjoy the experience overall.

Another benefit of working with a mainstream platform is the ability to leverage and adapt the work of a large community which helps keep our development and operations costs down. Over 8,400 individuals have contributed code to OpenStack, with 1,000-2,500 contributors participating in each major release [28]. Leveraging their contributions, over the lifetime of the project we were able to offer our community new key features such as whole disk image boot, multi-tenant networking, serial console integration, support for non-x86 architectures, and user-customizable firewalls—simply by upgrading to a new OpenStack version. Future capabilities already possible due to upstream contributions include self-service BIOS customization and detachable remote storage; both have been common user requests. From the operator's perspective, deploying and managing Chameleon was made simpler and more reliable by the Kolla project, which provides a packaging of OpenStack as Docker containers [29] and a set of highly-configurable Ansible provisioning scripts [30] to orchestrate the setup and maintenance of the deployment. Further, ~6,000 individuals have been involved in reviewing all code changes [31]: thus, by using a mainstream infrastructure we also benefit from a built-in large-scale quality control mechanism. No less valuable has been the access to the existing documentation and support systems within the community: the openstack-discuss mailing list [32] sees between 500-1000 messages each month, the OpenStack Q&A forum [33] has over 18,000 answered questions, and on many occasions we were able to get a workarounds or patches for bugs within days simply by filing a ticket to the official tracker.

The flip side of leveraging contributions of others is the opportunity to contribute to and shape a mainstream infrastructure. On a practical level, this magnifies our investment in the infrastructure and our broader impacts as any new features and additions we make to OpenStack are also impacting communities beyond the testbed. Because its hardware resources have always been constrained relative to the demands of its user community, Chameleon required a system for allocating and managing resources (including advance reservations), which our team implemented by reviving and significantly improving the OpenStack Blazar project. This attracted the interest of others and we were subsequently able to partner on development with contributors from NTT (Japan) and DOCOMO Euro-Labs (Germany) who use the component in Software Defined Networking applications. As a result of this collaboration, Blazar became an official top-level OpenStack component in 09/17 and has been included in each OpenStack releases since Queens. Other significant new features developed by the Chameleon team include bare metal snapshotting and enablement of slice creation via integration with ExoGENI stitching implementation; in addition, we made many smaller contributions in the form of bug fixes and patches.

These advantages are offset by some costs. Since it solves a complex problem, OpenStack is complex; thus operating, and in particular extending it, requires both development skills and deep expertise in the underlying systems and concepts, putting pressure on operator hours and level of skill. The most problematic manifestation of this complexity in our experience used to be OpenStack upgrades; this has improved significantly with tools that aid operators in these tasks, such as the aforementioned Kolla project. The size and structure of the open source community imposes its own overhead and requires commitments both in process and in time: patches must be reviewed, meetings must be attended, changes must be formally proposed and approved, and documentation and tests must be written. These commitments are the price of admission to an open-source community that, in our experience, ultimately returns the investment many times over in the form of support, debugging, development, and partnership.

Looking beyond practical benefits, building on top of a mainstream infrastructure helps settle a point of intellectual interest in that it provides a direct answer to the question: can clouds support CS system experimentation? While different clouds will of course be configured differently, Chameleon represents a configuration that satisfies this condition; we describe this configuration in this paper but it is also expressed in practical form via code, recipes, and settings that are publicly available and suitable for replication whether in academia or commercially. Like the Chameleon interfaces we support, that recipe has multiple "entry points": users may elect to simply install OpenStack with our contributions—they may elect to replicate the Chameleon configuration in every detail—or they may choose something in between. We facilitate this by providing a packaging of Chameleon Infrastructure (CHI) that contains not only OpenStack but also extensions including Grid'5000 and ExoGENI additions, as well as an operational model we developed that makes clouds of this type cost-effective to provide. This packaging, called CHI-in-a-Box [34], has recently been installed at Northwestern University and augmented Chameleon capabilities by providing modest but unique networking hardware.

5 Fostering Replicability and Sharing

Perhaps the most important lesson learned from Chameleon is that testbeds provide not only a platform for instruments but also generate shareable digital artifacts such as images, orchestration templates, datasets, tools, notebooks, and others. Those artifacts typically represent either a complete experiment or an important part of one and can be used to reenact it on the testbed on which it was created. For example, over the lifetime of the Chameleon testbed, users created 120,000 disk images and 31,000 orchestration templates that can be used for such purpose. This presents an opportunity: since the generated artifacts can be used to repeat experiments, sharing them should allow others to repeat experiments, introduce variation, or extend experiments more easily. We posit that fostering that sharing will contribute to the overall goal of providing a scientific instrument to advance CS systems research by both reducing time to discovery and providing a more fertile ground for sharing of ideas. The question arises how specifically experiments should be represented and structured—and then how specifically they should be shared.

Chameleon has supported a variety of mechanisms to aid repeatability over its lifetime. First, the Chameleon hardware is versioned, which allows users to easily identify any changes which would introduce variation. Users can also version the images they configure, and publish them in a catalog. Since this still requires a user to keep track of which appliances were deployed on which testbed version, we introduced a system, called Chameleon's Experiment Précis [35], which captures all the distributed events generated by a user in the testbed, and presents him or her with a summary (a précis) of their experiment. Then, working with an accurate and impartial record of their work, the user can filter or preview the events to include only the relevant ones. The précis data can be used to generate a description of the experiment in English, or potentially an actionable description of the experiment in the form of orchestration templates or commands that will reproduce the experiment. Generating Heat orchestration templates in this way is in fact the objective of an OpenStack Flame [36] tool. Overall, the Experiment Précis is somewhat analogous to a shell's "history" command with the critical difference that it captures distributed rather than local events – though its output is less "actionable".

This raises the question of what an actionable representation of an experiment should ideally look like. Orchestration systems such as profiles in CloudLab or Heat in Chameleon require adhering to a strict machine-readable syntax, often formulating that syntax in a declarative text file, or providing a layer of indirection that allows the user to work in a higher-order language. Furthermore, these systems are transactional, either fulfilling the topology or not, making complex configurations difficult to develop and iterate upon. Proposed workflow systems [37] experience similar challenges [38]. They fundamental problem from the user perspective in these cases is that a user must invest extra time to make an experiment reproducible. This leads to the "reproducibility dilemma": the user needs to choose whether to invest time into making an experiment replicable or continuing with other research. Ideally, a system that represents an experiment would allow the experimenter to develop it gradually and interactively reflecting the often meandering creative process, support experimental "story-telling" for human as opposed to just machine users, and—true to our philosophy—be an open source project in mainstream use.

In researching solutions to this problem, we considered computational notebooks such as Wolfram Mathematica [39] and Jupyter Notebooks [40], which combine expository text, executable code, and presentation of results in one human-readable, interactive document. Jupyter's newfound ubiquity across the research landscape and its extensible architecture provided fertile ground for exploration; to leverage it, we integrated Jupyter and Chameleon with the intent to reduce the gap between designing an experiment and sharing it. As a result, users can log in to Chameleon Jupyter server with their testbed credentials; these credentials are implicitly bound to the user's isolated Jupyter Notebook server, allowing the user to call Chameleon APIs (via the CLI interface or Python APIs) directly within their notebook's code blocks. This allows a notebook to completely re-create a pre-existing testbed topology. We additionally mounted Chameleon's object store as a virtual drive in the Jupyter application to allow shared storage and therefore collaboration between users. The user's Jupyter server comes pre-installed with several libraries that aid interfacing with an experiment on Chameleon [41] (e.g., creating a lease and launching an instance and then executing commands remotely via SSH). As we've improved the Jupyter interface to Chameleon it has seen an increasing share of testbed usage; over the last year 10% of our monthly active users have been using Jupyter as their interface to the system.

While the notebook may be an appropriate way to package an experiment without disrupting the flow of research, the challenge of properly disseminating and discovering these artifacts remains. At the end of 2019, we implemented the first version of a Sharing Portal [42] to allow users to publish and discover these notebook-based artifacts. Users can publish a set of files directly from the Jupyter interface via a custom UI extension; the files are compacted into an archive and published to CERN's Zenodo [43] for long-term storage, where they are also assigned a DOI for citation. The Sharing Portal then maintains a reference to the published artifact along with helpful metadata such as tags and documentation. Other Chameleon users can search for artifacts within the portal and "re-play" them on Chameleon with one click: an ephemeral Jupyter server tied to the artifact is dynamically provisioned, including additional software dependencies defined by the publisher. Users can version their artifacts by publishing a new set of files and creating a new version (and DOI) on Zenodo. Though too early in its lifetime to provide a quantitative analysis of its impact, we expect to continue investing in this area going forward.

6 Related work

Some of our design decisions in Chameleon were informed by our earlier work on experimental testbeds including FutureGrid [44,45] and ExoGENI [8], as well as our long standing close collaboration with Grid'5000 [1]. In particular, we are indebted to the latter two projects for not only providing insight but also specific capabilities that were directly integrated into the testbed; the stitching capabilities in the case of ExoGENI and the resource representation and related tooling in the case of Grid'5000. At the same time, Chameleon represents a significantly different approach from any of them in many ways, most prominently in that it is configured for cloud computing research (unlike ExoGENI), supports bare metal reconfiguration (unlike FutureGrid), and is based on a mainstream infrastructure (unlike Grid'5000).

We additionally leverage the experience gained by the rich history of CS testbeds, ranging from those specializing in networking (PlanetLab [46], GENI [4], Emulab [2], OneLab [47], CENI [48]) and wireless/IoT (ORBIT [49], FIT [50], CityLab [51]) to systems (CloudLab [3]) and security (DeterLab [52]); Chameleon complements these systems in that it largely focuses on a different area of research and thus supports different types of experiments. The most similar testbed to Chameleon is CloudLab (itself another NSFFutureCloud testbed). We differ from CloudLab primarily in specific design decisions we made in building the system, many of which are described in this paper as well as in our early emphasis on reproducibility and sharing by integration of tools like Experiment Précis and Jupyter notebooks. The most significant difference however is that we built Chameleon on top of a mainstream cloud infrastructure for reasons described above.

Clouds are being increasingly used in science and many of them elect to use OpenStack, e.g., Jetstream [53], Aristotle [54], Comet [55], Bridges [56], and NeCTAR [22] all represent different configurations of the system. The most significant difference from Chameleon is that these are relatively standard cloud configurations, designed primarily to support domain science applications rather than CS experimentation, and differ from Chameleon significantly on key features such as bare metal reconfiguration. However, there are significant commonalities on the operations side ensuring that we are able to leverage their contributions, with the OpenStack SIG being one avenue of communication. An interesting recent addition is the CloudBank project [57], which will provide tools, training, and credits for CS research on virtualized commercial clouds; since we provide similar services via our KVM cloud, we look forward to working with this project.

Reproducibility of CS experiments [58] is another area in which our contributions relate to other work in the field. Several projects have used Jupyter notebooks as a mechanism for encapsulating and reproducing research [59, 60]. Managed "Notebook-as-a-Service" platforms e.g., CodeOcean [61], WholeTale [62], and Nextjournal [63] have further elevated the profile and utility of Jupyter for this purpose. Workflow solutions such as Popper [37] aim to aid reproducibility in a different way and are more relevant to our efforts on Experiment Précis. Chameleon differs from all of these examples in that it integrates reproducibility tools in the context of a testbed, allowing users to leverage a commonality of platform to replicate not only the process of experimentation, but also the requisite hardware configurations and topologies.

7 Conclusions

Chameleon represents a unique testbed in that it expresses the capabilities needed for CS research in terms of mainstream cloud functionality. This is an important step to understand how such capabilities may be supported more ubiquitously, with more discernment, and in a more cost-effective manner. This paper discusses the specific design decisions, extensions, and configurations that we chose in order to do so, and evaluates them within a framework that seeks to establish how they influenced the set of supported experiments and how they influenced the community of users the testbed was able to support. Our contribution is accompanied by software that was contributed or integrated with a mainstream open-source cloud implementation (OpenStack) so that a cloud of this type – or its evolutions or variations – can be supported by anyone.

The most important part of our experience is the insight that though originally created to support the most experiments for the most experimenters, testbeds have also become both a generator and an essential platform for sharing digital representations of experiments. While our understanding of digital sharing ecosystem still evolves, and is likely to evolve for some time, we proposed some approaches that our user community has found useful and we look forward to contributing to this area in the future.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

Availability

Traces from Chameleon CHI and KVM have been publically available at [64, 65] for the last couple of years and used in a variety of resource management publications. Not all the data used in this paper is reflected in those traces though we are currently discussing revisions to both the content and format. All the code described here is open source.

References

[1] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. In I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[2] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[3] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, 2019.

[4] M. Berman, J.S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: A Federated Testbed for Innovative Network

Experiments. *Computer Networks*, 61:5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[5] Chameleon Resource Discovery. https://www.chameleoncloud.org/hardware/.

[6] M. Cevik, P. Ruth, K. Keahey, and P. Riteau. Wide-area Software Defined Networking Experiments using Chameleon. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 811–816. IEEE, 2019.

[7] OpenStack. https://www.openstack.org/.

[8] I. Baldin, J.S. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, and J. Mills. ExoGENI: A Multi-Domain Infrastructure-as-a-Service Testbed. In McGeer et al. [4], pages 279–315. Special issue on Future Internet Testbeds – Part I.

[9] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth. Chameleon: a Scalable Production Testbed for Computer Science Research. In Jeffrey Vetter, editor, *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 3 of *Chapman & Hall/CRC Computational Science*, chapter 5, pages 123–148. CRC Press, Boca Raton, FL, 1 edition, May 2019.

[10] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, 2018.

[11] Using New vCPU-based On-demand Instance Limits with Amazon EC2. https://aws.amazon.com/blogs/compute/preview-vcpu-based-instance-limits/.

[12] K. Keahey, P. Riteau, J. Anderson, and Z. Zhen. Managing Allocatable Resources. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 41–49, July 2019.

[13] C.P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, M. Rosenblum, et al. Virtual Appliances for Deploying and Maintaining Software. In *LISA*, volume 3, pages 181–194, 2003.

[14] OpenStack Heat. https://docs.openstack.org/heat/latest.

[15] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *2008 IEEE Fourth International Conference on eScience*, pages 301–308. IEEE, 2008.

[16] Chameleon Automated Deployment. https://chameleoncloud.readthedocs.io/en/latest/technical/complex.html#automated-deployment.

[17] Prometheus. https://prometheus.io/.

[18] K. Keahey, J. Anderson, P. Ruth, J. Colleran, C. Hammock, J. Stubbs, and Z. Zhen. Operational Lessons from Chameleon. In *Proceedings of the Humans in the Loop: Enabling and Facilitating Research on Cloud Computing*, pages 1–7. 2019.

[19] OpenStack Ironic. https://docs.openstack.org/ironic/latest.

[20] The 2018 OpenStack User Survey Report. https://www.openstack.org/user-survey/2018-user-survey-report/.

[21] OpenStack Operator Spotlight: CERN. https://superuser.openstack.org/articles/openstack-operator-spotlight-cern/.

[22] NeCTAR Cloud. https://nectar.org.au/research-cloud/.

[23] NASA's JPL powers planetary exploration with Red Hat OpenStack platform. https://www.redhat.com/en/about/press-releases/nasa%E2%80%99s-jet-propulsion-laboratory-powers-planetary-exploration-red-hat-openstack-platform.

[24] OpenStack Scientific SIG. https://wiki.openstack.org/wiki/Scientific_SIG.

[25] OpenStack Scientific Working Group Launches at OpenStack Summit Austin. https://superuser.openstack.org/articles/openstack-scientific-working-group-launches-at-openstack-summit-austin/.

[26] Mirantis Partners with OpenStack Foundation to Support Upgraded COA Exam. https://www.globenewswire.com/news-release/2019/10/17/1931470/0/en/Mirantis-Partners-With-OpenStack-Foundation-to-Support-Upgraded-COA-Exam.html.

[27] Net Promoter Score. https://www.netpromoter.com/know/.

[28] Stackalytics: Total Commits. https://www.stackalytics.com/?metric=commits&release=all.

[29] Kolla. https://docs.openstack.org/kolla/latest/.

[30] Kolla-ansible. https://docs.openstack.org/kolla-ansible/latest/.

[31] Stackalytics: Total Reviews. https://www.stackalytics.com/?metric=marks&release=all.

[32] Openstack-discuss Mailing List Archives. http://lists.openstack.org/pipermail/openstack-discuss/.

[33] Ask OpenStack. https://ask.openstack.org.

[34] CHI-in-a-Box. https://github.com/ChameleonCloud/chi-in-a-box.

[35] S. Wang, Z. Zhen, J. Anderson, and K. Keahey. Reproducibility as Side Effect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18 Poster)*. IEEE Press, 2018.

[36] OpenStack Flame. https://opendev.org/x/flame.

[37] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570. IEEE, 2017.

[38] M.A. Sevilla and C. Maltzahn. Popper Pitfalls: Experiences Following a Reproducibility Convention. In *Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems*, page 4. ACM, 2018.

[39] S. Wolfram. The Mathematica Book. *Assembly Automation*, 1999.

[40] Project Jupyter. https://jupyter.org/.

[41] Python-chi: Chameleon Python library. https://github.com/chameleoncloud/python-chi.

[42] M. King, J. Anderson, and K. Keahey. Sharing and Replicability of Notebook-Based Research on Open Testbeds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19 Poster). IEEE Press*, 2019.

[43] Zenodo. https://zenodo.org.

[44] G.C. Fox, G. von Laszewski, J. Diaz, K. Keahey, J. Fortes, R. Figueiredo, S. Smallen, W. Smith, and A. Grimshaw. Futuregrid: a Reconfigurable Testbed for Cloud, HPC, and Grid Computing. In *Contemporary High Performance Computing*, pages 603–635. Chapman and Hall/CRC, 2017.

[45] G. von Laszewski, G.C. Fox, F. Wang, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J Vöckler, R.J. Figueiredo, J. Fortes, et al. Design of the Futuregrid Experiment Management Framework. In *2010 Gateway Computing Environments Workshop (GCE)*, pages 1–10. IEEE, 2010.

[46] L. Peterson, A. Bavier, M. E Fiuczynski, and S. Muir. Experiences Building Planetlab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366, 2006.

[47] S. Fdida, T. Friedman, and T. Parmentelat. OneLab: An Open Federated Facility for Experimentally Driven Future Internet Research. In *New Network Architectures*, pages 141–152. Springer, 2010.

[48] Y. Xu, X. Lu, and Y. Zhang. The Development of China's Next Generation Network and National Service Testbed. *development*, 14:3, 2015.

[49] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-generation Wireless Network Protocols. In *IEEE Wireless Communications and Networking Conference, 2005*, volume 3, pages 1664–1669. IEEE, 2005.

[50] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, et al. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464. IEEE, 2015.

[51] J. Struye, B. Braem, S. Latré, and J. Marquez-Barja. CityLab: A Flexible Large-scale Multi-technology Wireless Smartcity Testbed. In *Proceedings of the 27th European Conference on Networks and Communications (EUCNC), 18-21 June 2018, Ljubljana, Slovenia*, pages 374–375, 2018.

[52] J. Mirkovic and T. Benzel. Teaching Cybersecurity with DeterLab. *IEEE Security & Privacy*, 10(1):73–76, 2012.

[53] C.A. Stewart, T.M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, et al. Jetstream: a Self-provisioned, Scalable Science and Engineering Cloud Environment. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 29. ACM, 2015.

[54] R. Knepper, S. Mehringer, A. Brazier, B. Barker, and R. Reynolds. Red Cloud and Aristotle: Campus Clouds and Federations. In *Proceedings of the Humans in the Loop: Enabling and Facilitating Research on Cloud Computing*, pages 1–6. 2019.

[55] H. Kim and M. Parashar. CometCloud: An Autonomic Cloud Engine. *Cloud Computing: Principles and Paradigms*, pages 275–297, 2011.

[56] S. Bridges. Cancer Genomics Cloud.(June 2017). *Retrieved June*, 1:2017, 2017.

[57] CloudBank. https://www.cloudbank.org/.

[58] D.G. Feitelson. From Repeatability to Reproducibility and Corroboration. *ACM SIGOPS Operating Systems Review*, 49(1):3–11, 2015.

[59] S.R. Piccolo and M.B. Frampton. Tools and Techniques for Computational Reproducibility. *GigaScience*, 5(1):30, 2016.

[60] T. Kluyver, B. Ragan-Kelley, F. Pérez, B.E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J.B. Hamrick, J. Grout, S. Corlay, et al. Jupyter Notebooks - a Publishing Format for Reproducible Computational Workflows. In *ELPUB*, pages 87–90, 2016.

[61] Code Ocean. https://codeocean.com/.

[62] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M.B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B.D. Mecum, J. Nabrzyski, et al. Computing environments for reproducibility: Capturing the “Whole Tale”. *Future Generation Computer Systems*, 94:854–867, 2019.

[63] Nextjournal. https://nextjournal.com.

[64] Chameleon CHI Traces, November 2019. https://zenodo.org/record/3709794#.XsgpCRNKiL8.

[65] Chameleon KVM Traces, November 2019. https://zenodo.org/record/3709958#.XsgoqRNKiL8.

SPINFER: Inferring Semantic Patches for the Linux Kernel

Lucas Serrano
Sorbonne University/Inria/LIP6

Van-Anh Nguyen
Sorbonne University/Inria/LIP6

Ferdian Thung
School of Information System
Singapore Management University

Lingxiao Jiang
School of Information System
Singapore Management University

David Lo
School of Information System
Singapore Management University

Julia Lawall, Gilles Muller
Inria/Sorbonne University/LIP6

Abstract

In a large software system such as the Linux kernel, there is a continual need for large-scale changes across many source files, triggered by new needs or refined design decisions. In this paper, we propose to ease such changes by suggesting transformation rules to developers, inferred automatically from a collection of examples. Our approach can help automate large-scale changes as well as help understand existing large-scale changes, by highlighting the various cases that the developer who performed the changes has taken into account. We have implemented our approach as a tool, Spinfer. We evaluate Spinfer on a range of challenging large-scale changes from the Linux kernel and obtain rules that achieve 86% precision and 69% recall on average.

1 Introduction

The Linux kernel is present today in all kinds of computing environments, from smartphones to supercomputers, including both the latest hardware and "ancient" systems. This multiplicity of targets with widely varying needs has implied that the code base has grown steadily over the Linux kernel's 28-year history, reaching 18M lines of code in Linux v5.4 (Nov. 2019). The complexity of designing an operating system kernel and the need to continually take into account new requirements has implied that the design of internal interfaces must be revisited, triggering the need for repetitive changes among the users of these interfaces that may affect an entire subsystem, or even the entire source tree. The size and number of the needed changes can discourage developers from performing them, and can introduce errors. Furthermore, when code is incompletely transformed, the volume of the performed changes can obscure the conditions that later developers who want to complete the change should take into account.

Since 2008, the automatic C code transformation tool Coccinelle [12] has been part of the Linux kernel developer toolbox for automating large-scale changes in kernel code. Coccinelle provides a notion of semantic patch allowing kernel developers to write transformation rules using a patch-like [16] (i.e., diff-like) syntax, enhanced with metavariables to represent common but unspecified subterms and notation for reasoning about control-flow paths. Given a semantic patch, Coccinelle applies the rules automatically across the code base. Today, Coccinelle is widely adopted by the Linux community: semantic patches are part of the Linux kernel source tree, are invokable from the kernel build infrastructure, and are regularly run by Intel's Linux kernel 0-day build-testing service [10]. Semantic patches have been written by kernel developers to make timers more secure [5], prepare the Linux kernel for the overflow of 32-bit time values in 2038 [6], reduce code size [27], etc.

Kernel developers use Coccinelle by first writing a semantic patch to perform a desired change, then applying it to the code base using Coccinelle and manually checking the resulting patch, and finally submitting the resulting patch for review and integration, according to the standard kernel development process [28]. Semantic patches have also been recognized as providing a precise means of communication about changes; developers include semantic patches in the commit logs of large-scale changes, and maintainers ask for them if they are not present, showing that semantic patches are considered to be valuable in the review and subsequent maintenance process. Still, there remain large-scale changes in the Linux kernel commit history where Coccinelle has not been used. Through discussions with Linux kernel developers we have learned that some know that Coccinelle would be helpful to them but, as they do not use it often, they do not remember the syntax. They also report that this realization often comes after performing a few manual changes. Furthermore, Coccinelle does not help developers understand existing large-scale changes, if no semantic patch is provided.

To better help developers, we propose to *infer semantic patches from existing change examples*, represented by a collection of files from before and after a change has been applied. Semantic patch inference can help developers understand previous changes without looking at hundreds of change instances. Semantic patch inference can also help developers with little Coccinelle knowledge use semantic patches if they are willing to make a few instances of the change manually.

Inferring semantic patches from real Linux kernel code changes, however, raises some challenges. The Linux kernel is written in C, which is a low-level language. Individual functionalities may be implemented as multiple C statements connected by control-flow and data-flow constraints. These constraints must be captured from the change examples and validated in the code to transform. A single kernel interface may expose multiple functions and data structures that can be used in various ways, any or all of which may be affected by a change. Inference must thus be able to cope with multiple change variants. Finally changes that are relevant to many files may be tangled with *noise*, *i.e.*, changes that are specific to a particular file, and thus are not worth automating. Semantic patch inference should be able to ignore such changes.

In this paper, we propose an approach to semantic patch inference that scales to the needs of systems code, such as the Linux kernel. Starting with the intraprocedural control-flow graphs of the original and changed functions found in the examples, our approach iteratively identifies code fragments having a common structure and common control-flow relationships across the examples, and merges them into a rule proposition. During this iterative merging process, rules are split as inconsistencies appear. Our approach is able to infer semantic patches from examples that overlap, that implement a family of transformations, and that may include noise. We have implemented our approach as a tool, *Spinfer*, targeting semantic-patch inference for the Linux kernel. Spinfer-inferred semantic patches can be read and understood, reviewed, and automatically applied to the Linux kernel.

The contributions of this paper are as follows:

- We propose a taxonomy of challenges that must be handled by transformation-rule inference tools and assess recent work and our proposed approach according to this taxonomy.
- We propose an approach to automatic inference of semantic patches that takes control and data-flow, multiple change instances, multiple change variants and unrelated changes into account. We provide an implementation of this approach for C code in the tool Spinfer.
- We evaluate Spinfer on a large set of 80 changes, affecting thousands of files drawn from recent Linux kernel versions, against semantic patches written by a Coccinelle expert. Generated semantic patches achieve on average 86% precision and 69% recall.

The rest of this paper is organized as follows. Section 2 presents some motivation for our work, our taxonomy and the most closely related work. Section 3 presents our approach and its implementation in Spinfer. Section 4 evaluates Spinfer on 80 recent sets of changes to the Linux kernel. Finally, Section 5 presents other related work and Section 6 concludes.

2 Background and Motivation

We first present an example of a large-scale change from the Linux kernel development history. Based on the challenges identified in this example, we then propose a taxonomy of challenges for transformation-rule inference. We then assess related approaches in terms of this taxonomy.

2.1 Motivating example

Linux kernel timers were originally initialized by a multi-step process, involving a call to **`init_timer`** to initialize a timer structure and two assignments to initialize the fields **`function`** and **`data`**, describing what to do when the timer expires. In Linux 2.6.15, released in 2006, **`setup_timer`** was introduced to combine these operations and thus simplify the source code. Elimination of **`init_timer`** got off to a slow start, with 73 changes in 2008 and then little activity until 2014. In 2014-2016 there were 43, 93, and 37 changes per year, respectively. The remaining 267 calls were removed in 2017, when it was also recognized that incremental initialization using **`init_timer`** represents a security hole.[1]

Figure 1 illustrates some instances of the **`init_timer`** change.[2] We examine these instances in terms of the challenges they pose for semantic patch inference.

Control-flow. The change typically involves removing three statements, *i.e.*, the call and the two field initializations. These three statements are typically part of some larger block of code, and thus do not correspond to an independent subtree of an Abstract Syntax Tree (AST). They are not always contiguous either (**`nicstar.c`**).

Data-flow. The change involves keeping the same expression for the **`init_timer`** and **`setup_timer`** first argument, and for the structure used in the **`data`** and **`function`** field initialization.

Multiple variants. While the examples basically perform the same operations, at a detailed level there are a number of variations. For example, the first two examples initialize both the **`function`** and **`data`** fields, but in the opposite order. Semantic-patch inference should be able to proceed in the face of these variants and generate the needed rules. The last example presents yet another variant that does not initialize the **`data`** field at all, retaining the default value of 0. This variant poses a further challenge, as it overlaps with the other variants, in that all of the examples remove an **`init_timer`** call and the initialization of the **`function`** field. Semantic-patch inference has to *carefully order* the resulting rules such that a rule setting the third argument of **`setup_timer`**, representing the timer data, to **`0UL`** does not apply to code where a value for the **`data`** field is specified.

[1] https://lkml.org/lkml/2017/8/16/817

[2] Commit b9eaf1872222. All commits cited in this paper are available at https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

drivers/s390/block/dasd.c

```
- init_timer(&device->timer);
- device->timer.function = dasd_device_timeout;
- device->timer.data = (unsigned long) device;
+ setup_timer(&device->timer, dasd_device_timeout,
+         (unsigned long)device);
```

drivers/atm/nicstar.c

```
- init_timer(&ns_timer);
+ setup_timer(&ns_timer, ns_poll, 0UL);
  ns_timer.expires = jiffies + NS_POLL_PERIOD;
- ns_timer.data = 0UL;
- ns_timer.function = ns_poll;
```

drivers/net/wireless/intersil/hostap/hostap_hw.c

```
- init_timer(&local->passive_scan_timer);
- local->passive_scan_timer.data =
-         (unsigned long) local;
- local->passive_scan_timer.function =
-         hostap_passive_scan;
-
- init_timer(&local->tick_timer);
- local->tick_timer.data = (unsigned long) local;
- local->tick_timer.function = hostap_tick_timer;
+ setup_timer(&local->passive_scan_timer,
+         hostap_passive_scan, (unsigned long)local);
+ setup_timer(&local->tick_timer, hostap_tick_timer,
+         (unsigned long)local);
```

arch/blackfin/kernel/nmi.c

```
- init_timer(&ntimer);
- ntimer.function = nmi_wdt_timer;
+ setup_timer(&ntimer, nmi_wdt_timer, 0UL);
```

Figure 1: Variants of the **`init_timer`** change

Multiple instances. Another form of variation in the examples is the number of instances of a given change. Most of the examples initialize only one timer, but **`hostap_hw.c`** initializes two. To concisely describe the overall change, it is better to obtain a rule for a single timer that applies to this code twice, rather than obtaining a rule specific for this case.

Noise. Change examples can also contain extraneous changes from which it is not useful to infer semantic patches. A single patch may, for example, change a function definition and update the uses of the function accordingly. The change to the definition is a one-time operation, so it is not useful to automate. Another possibility is the presence of other changes, such as minor refactorings. While research on application software has found that the latter tangled changes are frequent [9], the Linux kernel documentation requires that developers separate their changes into one change per patch [28], and thus we expect tangled changes to be a minor issue in our setting in practice.

2.2 Taxonomy

Based on the above examples and study of other large-scale changes in the Linux kernel, we have created a taxonomy, shown in Table 1, of challenges for transformation-rule inference. This taxonomy can be used to characterize particular change examples and to compare transformation-rule inference approaches.

Table 1: Taxonomy of challenges for transformation inference

***C*: Control-flow dependencies**
0. *No* control-flow dependencies between changed terms
1. *Intraprocedural dependencies* between changed terms
2. Intraprocedural dependencies between changed and *unchanged terms*
3. Some terms must *not* appear within relevant control-flow paths
4. *Interprocedural* control-flow dependencies between changed and unchanged terms
***D*: Data-flow dependencies**
0. *No* data-flow dependencies between changed terms
1. Data-flow dependencies between changed terms
2. Data-flow dependencies between changed and unchanged terms
***I*: Change instances per function**
0. One instance
1. Multiple instances
2. Overlapping instances
***V*: Change variants**
0. One variant
1. Multiple variants
2. Multiple variants with specific order
***N*: Noise (errors and unrelated changes)**
0. *No* noise
1. Contains noise

The taxonomy considers the relationship between changed terms in a single change instance (C and D), the ways in which multiple change instances can appear within a single function (I), and the possibility of change variants (V) and unrelated changes (N, noise). For control-flow dependencies, the taxonomy entries range from no dependencies (C_0) to changes that requires intraprocedural dependencies to changed terms (C_1) to changes that involve dependencies between both changed and unchanged terms across multiple functions (C_4). For data-flow dependencies, the taxonomy entries range from no dependencies (D_0) to data dependencies between both changed and unchanged terms (D_2). For multiple change instances, the taxonomy entries range from one instance per function (I_0) to overlapping instances (I_2). For multiple variants, the taxonomy entries range from one variant at all change instances (V_0) to multiple variants that have to be considered in a specific order (V_2). For noise, the taxonomy entries have either the absence of noise (N_0) or presence of noise (N_1). Spinfer targets C_1, D_2, I_2, V_2, N_1 (control-flow and data-flow constraints, potentially overlapping change instances, multiple order-dependent change variants, and the possibility of noise in the examples).

2.3 Existing tools

A number of tools have previously been developed to infer transformation rules or update API uses automatically. Most of these tools target user-level applications written in object-oriented languages, typically Java, while we target an operating system kernel written in C. While these different kinds of code bases raise different challenges, all of the issues we identify with existing tools also apply to our setting.

Individual examples. Sydit [17], A4 [11], Meditor [30], AppEvolve [7], and GenPat [8] generate transformation rules from individual change examples. All of these approaches except GenPat abstract over the examples by abstracting over variables. In practice, change examples mix generic computations and computations that are specific to a particular application. Abstracting over variables is not always sufficient to abstract away these application-specific computations. Such approaches thus obtain low recall on anything but the simplest examples. GenPat [8] bases abstraction decisions on the frequency of various terms in code repositories such as GitHub. As GenPat's abstraction strategy is based on the properties of a large code corpus rather than on the change itself, it may infer transformations that are too generic, transforming code that should not be changed, thus reducing precision. The evaluation presented in the work on GenPat reflects these issues.

Approaches that abstract from individual examples may perform incorrect transformations if the examples are not given to the tool in the right order. For example, in the **`init_timer`** example, if **`nmi.c`** is provided as an example first, then all timer initialization code will be incorrectly updated with the data value 0, even if another data value is provided. AppEvolve addresses this issue by computing a common core of the provided examples, and attempting to apply the generated rules in order of their distance from this common core. The rule generated from the **`nmi.c`** example, however, is smaller than the others, and would thus be tested first by such a strategy. AppEvolve additionally proposes to use testing to validate the changed code, but the Linux kernel does not provide a comprehensive suite of easy-to-run high coverage test cases.

Several of these tools are also not able to identify change application sites. For example, Sydit requires the user to specify the affected function, while AppEvolve requires the user to specify the affected file name and line number. Manually specifying change sites is not practical for a code base the size of the Linux kernel.

In terms of the taxonomy, Sydit, A4, Meditor and GenPat target C_0, D_2, I_1, V_2, N_0; AppEvolve targets C_0, D_2, I_1, V_2, N_1 but requires preliminary manual rewriting of the change examples [29].

Multiple examples. Lase [18], Refazer [23], and Spdiff [1, 2] learn from multiple examples, identifying how to abstract over these examples based on the commonalities and differences between them.

Based on a set of change examples, Lase represents modifications as a sequence of AST edits and solves the Largest Common Subsequence Problem to find a transformation rule. This method implies that Lase cannot learn from examples containing unrelated changes or multiple variants, as we have seen for **`init_timer`**. Moreover these edit subsequences do not capture control-flow constraints and thus can generate incorrect changes, as illustrated below. In terms of the taxonomy Lase targets C_0, D_2, I_0, V_0, N_0

Refazer represents a transformation as a list of rewrite rules in a domain-specific language. It clusters changes from multiple examples and then infers one rule for each cluster. Like Lase this list of rewrite rules does not incorporate control-flow constraints and thus can generate incorrect changes. In terms of the taxonomy Refazer targets C_0, D_2, I_2, V_1, N_0.

Spdiff abstracts over common changes across the examples, incrementally extending a pattern until obtaining a rule that safely describes the complete change, taking control-flow constraints into account. It targets the Linux kernel and produces Coccinelle semantic patches. In terms of the taxonomy Spdiff targets C_2, D_2, I_2, V_0, N_0, however when tested on all of our examples it produced no correct results. For even simple function call replacements, we found that Spdiff spends much time on finding more complex but actually incorrect solutions.

We can note that, with the exception of Spdiff, all existing tools do not handle control-flow relationships between changed terms. To illustrate the impact of control flow, we have performed an experiment on Linux kernel-like code using Lase[3] (the distributed implementation of Refazer only supports restricted subsets of Python and C#, making it difficult to test in practice). Consider the following Linux kernel change example:[4]

```
- tport = kmalloc(sizeof(struct ti_port), GFP_KERNEL);
+ tport = kzalloc(sizeof(struct ti_port), GFP_KERNEL);
  if (tport == NULL) {
    dev_err(&dev->dev, "%s - out of memory\n", ...);
    status = -ENOMEM;
    goto free_tports;
  }
- memset(tport, 0, sizeof(struct ti_port));
```

Thus, from examples such as this one, Lase infers a rule that makes the following incorrect changes:

```
  if (a)
-   x = kmalloc(sizeof(*x), GFP_KERNEL);
+   x = kzalloc(sizeof(*x), GFP_KERNEL);
  else
-   memset(x, 0, sizeof(*x));
```

[3] Lase targets Java. We use C in the presented examples for consistency with the rest of the paper.

[4] Commit 7ac9da10af7f

and

```
- x = kmalloc(sizeof(*x), GFP_KERNEL);
+ y = kzalloc(sizeof(*y), GFP_KERNEL);
  y = kmalloc(sizeof(*x), GFP_KERNEL);
- memset(y, 0, sizeof(*y));
  memset(x, 0, sizeof(*x));
```

The first case shows that the inferred rule does not check that the **memset** affects the result of the call to **kmalloc** in the same control-flow path, thus performing an incorrect transformation. The second case shows that the inferred rule modifies the wrong **memset**, *i.e.*, the one affecting **y** rather than the one affecting **x**, and, worse, rearranges the variables so that **x** is no longer initialized. LASE only updates the first match in each function, and thus the second call to **kmalloc** is left unchanged. The former example may be unlikely for the specific functions of **kmalloc** and **memset**, but there is an instance analogous to the latter example in the Linux kernel history.[5]

Our approach. Given the difficulty of choosing an appropriate degree of abstraction from only one example, and given the availability of multiple examples of the use of most interfaces in the Linux kernel, our approach relies on multiple examples. Our approach is able to infer and express both control-flow and data-flow relationships that must be respected between fragments of code that are to be transformed. It adapts to variations in the examples to produce multiple rules and tolerates noise. Finally it produces transformation rules in a readable notation that is close to C code and is familiar to Linux kernel developers.

To the best of our knowledge, our approach is the only one capable of handling all these challenges at the same time and requires no rewriting of example code, making it suitable for systems code transformations.

3 Approach and Tool Design

In this section, we first give an overview of our approach, and then present the various steps used by our tool Spinfer to realize this approach. Spinfer is implemented as 10.6K lines of OCaml code. For parsing C code and mining control-flow relationships, it reuses the infrastructure of Coccinelle [12].

3.1 Overview

Our goal is to produce Coccinelle transformation rules based on a set of provided change examples, consisting of pairs of functions from *Before* and *After* the change. To motivate the steps of our approach, we first consider a semantic patch, composed of 4 rules, that a Coccinelle expert might write based on the **init_timer** examples illustrated in Section 2:

```
@@expression T,F,D;@@    @@expression T,F,D;@@
- init_timer(&T);         - init_timer(&T);
+ setup_timer(&T, F, D);+ setup_timer(&T, F, D);
  ...                       ...
- T.data = D;             - T.function = F;
- T.function = F;         - T.data = D;

@@expression T,F,D;@@    @@expression T,F;@@
- T.function = F;        - init_timer(&T);
- T.data = D;            + setup_timer(&T, F, 0UL);
  ...                      ...
- init_timer(&T);        - T.function = F;
+ setup_timer(&T, F, D);
```

The semantic patch consists of four transformation rules for the four different patterns of changes that occur in the examples. Each rule initially declares some *metavariables*, representing abstracted subterms, and then describes the changes as a sequence of lines of code patterns annotated with **-** and **+**, indicating code to remove and add, respectively. "..." represents a control-flow path, connecting the code matched by the pattern before the "..." to the pattern after the "...". Coccinelle applies a semantic patch to C source code in terms of the C source code's control-flow graph (CFG). A CFG as used by Coccinelle contains a node for each semicolon-terminated statement in the function, as well as for each if header, while header, etc. Coccinelle matches each rule of a semantic patch against the CFG of each function in a C file. When a match is found, Coccinelle transforms the code found in each matched CFG node according to the **-** and **+** annotations. More details about Coccinelle's semantic patch language SmPL are presented elsewhere [3].

We can observe that these four transformation rules share some constituents:

- Removal of a call to **init_timer**: **init_timer(&T);**
- Removal of a **data** field initialization: **T.data = D;**
- Removal of a **function** field initialization: **T.function = F;**
- Addition of the call **setup_timer(&T, F, D);**

We call these constituents *abstract fragments*, as they are fragments of a semantic change and each of them has a particular role in that change. Identifying a semantic patch as an assembly of small abstract fragments can help to solve the challenges highlighted in the introduction. In this view:

- Control-flow dependencies are the rules to assemble fragments together.
- Data-flow dependencies are the rules to match metavariables together.
- Variants are different assemblies of abstract fragments.
- Errors and noise are very unpopular abstract fragments.

[5] Commit 87755c0b3af6.

One can thus resolve these challenges by identifying the abstract fragments to consider and by determining how to assemble them together.

Following this observation, our approach focuses on finding abstract semantic patch fragments that will be assembled into one or more semantic patch rules. Each rule will match one of the variants illustrated by the examples. We have implemented this approach as a tool named Spinfer.

Starting from a set of *examples*, consisting of pairs of files before and after some changes, Spinfer constructs a set of transformation rules describing the changes. For this, Spinfer first identifies sets of common removed or added terms across the examples, then generalizes the terms in each set into a pattern that matches all of the terms in the set, and finally integrates these patterns into transformation rules that respect both the control-flow and data constraints exhibited by the examples, splitting the rules if necessary when inconsistencies appear.

Spinfer is organized as follows:

1. Identification of abstract fragments: Spinfer first clusters subterms from the examples having a similar structure and generalizes each cluster into an abstract fragment that matches all the terms in the cluster.

2. Assembling the rule-graphs: Spinfer then combines the abstracted fragments into a *semantic patch rule-graph*, a representation of a transformation rule as a graph, where nodes represent fragments to add and remove, and where edges are determined by control-flow dependencies exhibited in the associated examples.

3. Splitting: When assembling fails or when Spinfer detects data-flow inconsistencies, Spinfer splits existing rule-graphs into more specific ones.

4. Rule ordering: Finally, Spinfer orders the generated rules, removing redundant ones, to maximize precision and recall while minimizing the number of rules for the final semantic patch.

3.2 Identification of abstract fragments

The goal is to cluster nodes sharing similar subterms to form abstract fragments. Given that we have no a priori knowledge of the change variants illustrated by our examples, we must make an arbitrary decision about the granularity at which to investigate their subterms. Concretely, we choose the granularity of individual statements in a straightline code sequence, as well as function headers and headers of conditionals and iterators. An example from the **`init_timer`** code would be **`init_timer(&device->timer);`**. Such terms have the desirable property of being complete statements and expressions, with simple control-flow relationships between them. We refer to these terms as *nodes* as they will later correspond to the nodes of the control-flow graphs (CFGs) that we use to validate the control-flow constraints.

Spinfer then proceeds with an initial clustering of the CFG-nodes to be removed and added according to the examples. This clustering is independent of control-flow information, and is refined by control-flow constraints in the subsequent step. Each cluster will be represented by the smallest abstraction that matches all members in the cluster, known as the anti-unifier [21,22] of the cluster. Clustering code fragments using anti-unification has already been used in REVISAR [24] and CodeDigger [4]. However these approaches give the same weights to all anti-unifiers, regardless of their popularity, which cannot help to discriminate noise from relevant nodes. Spinfer overcomes this limitation by using techniques from text mining.

Node weighting: To facilitate clustering and noise detection, we want to give higher weight to anti-unifiers that are likely to form correct abstract fragments, and lower weight to anti-unifiers that are either too specific, manifested as rarely occurring across the set of examples, or too generic, manifested as occurring frequently within a single example, to describe the change. This goal is very similar to the goal of the *term frequency – inverse document frequency (TF-IDF)* [26] process used in text mining to highlight words that particularly characterize the meaning of a given document in a corpus. Spinfer requires the inverse notion, i.e., anti-unifiers that are common to many documents (i.e., functions), but do not occur too frequently in any given document.

Concretely, Spinfer uses a process that we call *function frequency – inverse term frequency (FF-ITF)*. In FF-ITF terms are anti-unifiers and a term weight increases with the number of functions that contains nodes matching this term (function frequency), and decreases with the number of nodes matching this term which appear in the same function (inverse term frequency).

The first step is to count how many times each anti-unifier appears. To do so, Spinfer first represents each node as a set of anti-unifiers that can match this node, from very abstract ones, that are likely to be shared by several nodes, to very concrete ones; we only consider anti-unifiers that at least abstract over local variables. Then given an anti-unifier $\mathcal{A}$, a set of functions F and a particular function $f \in F$ that contains the set of nodes N_f, our weight $w_{\mathcal{A},f}$ is:

$$\mathrm{FF}_{\mathcal{A}} = \frac{|\{f' \in F : \mathcal{A} \in f'\}|}{|F|}$$

$$\mathrm{ITF}_{\mathcal{A},f} = \log \frac{|N_f|}{|\{n \in N_f : \mathcal{A} \in n\}| + 1}$$

$$w_{\mathcal{A},f} = \mathrm{FF}_{\mathcal{A}} \times \mathrm{ITF}_{\mathcal{A},f}$$

This weight function is closely related to the one used in TF-IDF. We illustrate why this weight function works with an example on the **`init_timer(&ntimer);`** code fragment. We consider here only the 3 following anti-unifiers: (1) **`Expr;`**, (2) **`init_timer(&ntimer);`** and (3) **`init_timer(&Expr);`**. (1) is too generic and matches every statement, consequently it will have high function frequency but a very low inverse term frequency. Conversely (2) is too specific and probably matches only one node, so it will have a low function frequency but a high inverse term frequency. (3) matches **`init_timer`** function calls and nothing else, making it a good candidate to form correct abstract fragment. It will have both a high function frequency and a high inverse term frequency, which will result in a higher weight than (1) and (2).

Noise detection: The result of the node weighting gives a set of weighted anti-unifiers for every modified node of each example. The next step is to separate noise from relevant nodes. As noise is composed of very unpopular code fragments, all its anti-unifiers are either too generic or have very low function frequency, and as a consequence, have low weights. On the contrary relevant nodes have at least one anti-unifier with a high weight. Thus it is possible to distinguish noise nodes from relevant nodes by looking at the weight of their highest weighted anti-unifier.

To decide if a node is relevant, Spinfer compares the weight of the node's highest weighted anti-unifier to the average of the weights of the highest weighted anti-unifier of each node. If it is below a certain distance from this average, the node is considered to be noise. We have performed a separate experiment on the noise detection and looked for the distance at which we could mark nodes as noise without producing false positives. We found the ideal distance to be slightly less than 3 standard deviations.

Nodes marked as noise are dropped from further processing by Spinfer.

Clustering: In this step, we want to group together nodes that share a common high-weighted anti-unifier. For this Spinfer, proceeds with the clustering of the nodes not identified as noise. Spinfer first assigns to each node a characteristic vector encoding the weights of all anti-unifiers for this node. Our approach uses agglomerative hierarchical clustering with complete linkage, described by Zhao *et al.* [31], on our characteristic vectors. This approach has already been used for document classification in conjunction with TF-IDF weighting [31].

The number of clusters is determined using the best average *Silhouette score* [25], a score estimating the quality of the clustering, for all possible numbers of clusters. After this procedure we obtain groups of nodes that are very similar, and that will be transformed to abstract fragments in the next step.

Cluster (code and CFG-Diff node)	Abstraction

`init_timer(&device->timer);`
`init_timer(&ns_timer);`
	`init_timer(&X0);`

`device->timer.function =`
`    dasd_device_timeout;`
`ns_timer.function = ns_poll;`
	`X0.function = X1;`

`device->timer.data =`
`    (unsigned long) device;`
`ns_timer.data = 0UL;`
	`X0.data = X1;`

Figure 2: *Before* clusters and anti-unifiers from the **`init_timer`** examples. **`X0`** and **`X1`** are metavariables.

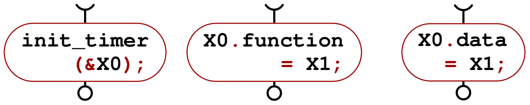


Figure 3: **`init_timer`** *Before* fragments

We illustrate this procedure with the **`init_timer`** change. In each of our **`init_timer`** examples, three nodes are removed and one node is added. The clusters are shown on the left side of Figure 2. Each element of a cluster is annotated with the example from which it comes and its position in that example.

For each cluster, Spinfer then create its anti-unifier, that retains the common parts of the terms in the cluster, and abstracts the subterms that are not common to all of the terms as *metavariables*, *i.e.*, elements that can match any term. The right side of Figure 2 shows the anti-unifier for each cluster.

Anti-unifiers represent sets of similar terms, but do not provide any control-flow information. In order to prepare for the next step, which constructs a semantic patch rule proposition based on control-flow constraints, Spinfer next expands each anti-unifier into the fragment of a control-flow graph that the anti-unifier's constituent node fragments represent, called an abstract fragment. An abstract fragment is composed of at least one node that contains the abstracted pattern, and a non-empty list of pairs of entry and exit points. Multi-node abstract fragments are created for complex control-flow structures such as conditionals and loops. Figure 3 shows some fragments for our **`init_timer`** example.

3.3 Assembling the semantic patch rule-graph

In order to address the first semantic patch inference challenge of capturing control-flow constraints, Spinfer relies on the notion of *dominance* [15] in CFGs. A node A *dominates* a node B in a directed graph $\mathcal{G}$ if every path from the entry node of $\mathcal{G}$ to B goes through A. A is then a dominator of B. Similarly, a node B *postdominates* a node A if every path from A to an exit node of $\mathcal{G}$ goes through B. B is then a postdominator of A. This notion generalizes straightforwardly to sets of nodes. Dominance characterizes the semantics of Coccinelle's "`...`" operator. A pattern of the form A ... B matches a fragment of C code if the set of terms matching

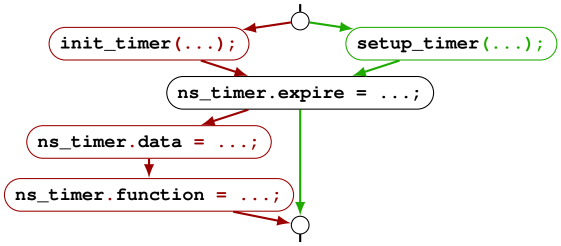


Figure 4: CFG-Diff for the `nicstar.c` `init_timer` change

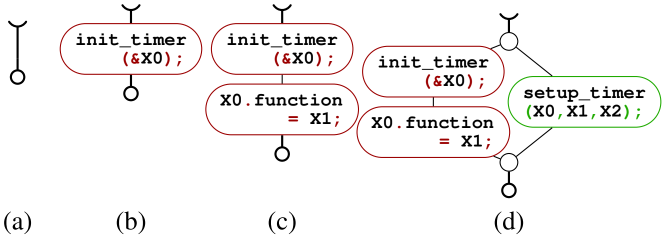


Figure 5: Steps in semantic patch-rule-graph construction

B postdominate the term matching *A*, meaning that from the code matching *A*, all outgoing paths reach code matching *B*.

Spinfer first constructs the control-flow graph (CFG) of each changed function in the examples, before and after the change. Each pair of before and after CFGs is then merged into a single CFG, referred to as a CFG-Diff. In the CFG-Diff, removed nodes from the before CFG are colored *Before* (red/-dark grey) and added nodes from the after CFG are colored *After* (green/light grey). An extract of the CFG-Diff for the `nicstar.c` change is shown in Figure 4.

To construct a semantic patch rule, Spinfer first constructs a *semantic patch rule-graph*, incrementally adding first *Before* fragments and then *After* fragments as long as the dominance relations in the semantic patch rule-graph respect the dominance relations in the example CFGs associated with the fragments.

We present the semantic patch-rule-graph construction algorithm in terms of the `init_timer` example. Spinfer starts with an empty semantic patch rule-graph, consisting of two special nodes, a global entry node and a global exit node, and a directed edge from the global entry node to the global exit node (Figure 5(a), the half-circle is the global entry node and the full circle is the global exit node). By convention, the global entry node dominates every node and every node postdominates it, and the global exit node postdominates every node and every node dominates it.

For our example, out of the available fragments (Figure 3), suppose that Spinfer first chooses the one representing the `init_timer` calls. This fragment can be added straightforwardly (Figure 5(b)) because the start node and the end node dominate and postdominate every node, respectively. Next, Spinfer may add the fragment representing the initialization of the timer's `function` field, below the node representing the `init_timer` call. Indeed, in both *Before* CFGs, the call to `init_timer` dominates the `function` field initialization, either because of a direct connection between them, or because of a control-flow path consisting of straightline code between them. The `function` field initialization likewise postdominates the `init_timer` call. The resulting semantic patch rule-graph is shown in Figure 5(c).

We defer the treatment of the initialization of the `data` field to Section 3.4, and turn to the integration of the *After* fragment, *i.e.*, the abstracted call to `setup_timer`. While the addition of the *Before* fragments relies on dominance relations between the fragment and the nodes already in the semantic patch rule-graph, this is not always possible for *After* nodes. Indeed, there exist no dominance relations between the *Before* and *After* nodes. When an addition is in fact a replacement, we observe that the *After* node and the *Before* node it is replacing share common context (non-modified) predecessors and successors (see Figure 4). Moreover, this context does dominate/postdominate both the deletion and the addition. Spinfer exploits this property to insert *After* fragments. For this, Spinfer replaces each entry and exit pair of the *After* fragment by the nearest context predecessors and successors, if any, and checks the domination properties on these context nodes instead. If the properties hold then the fragment is inserted. The context used is replaced by *Merge* nodes, that are omitted in the final semantic patch.

3.4 Splitting the semantic patch rule-graph

Assembling the semantic patch rule-graph can fail, due to inconsistencies. This situation occurs when there are multiple change variants in the examples, our third semantic patch inference challenge. One possible inconsistency is incompatible domination properties, representing inconsistent control flow. Another is the inability to map metavariables to terms in a way that allows the added code to be constructed from the removed code, representing a form of inconsistent data flow. Spinfer splits the semantic patch rule-graph and reduces the genericity of metavariables to address these issues.

Control-flow inconsistencies. Trying to insert the `data` field initialization fragment into the rule-graph shown in Figure 5(c), reveals a control-flow inconsistency: `dasd.c` initializes the `data` field after the `function` field, while `nicstar.c` initializes it before. To proceed, Spinfer chooses an element of the `data` field initialization cluster, say the one from `dasd.c`, and splits the cluster into one cluster of the instances that respect the same control-flow constraints and another cluster for the remaining instances. The latter is deferred for later integration. At the same time, Spinfer splits the semantic patch-rule graph into one copy derived from the graphs whose `data` field initialization respects the same control flow constraints as the `dasd.c` initialization and another copy for the rest. The latter rule-graph copy is likewise pushed onto a stack for later treatment.

After splitting the cluster and the semantic patch rule-graph, Spinfer can add the fragment representing `data` field initializations that are consistent with `dasd.c` into the rule-graph below the `function` field initialization, and then proceed to the integration of the *After* fragment, as described previously. With this rule-graph complete, Spinfer then proceeds to the next rule-graph on the stack and the remaining fragments.

A semantic patch rule-graph split may lead to a situation, as we have here, where a rule-graph represents only a single example. In this case, Spinfer leaves the generated semantic patch rule abstract, according to the metavariables motivated by the clustering, to allow the generated semantic patch rule to match code from other files. This extra abstraction, however, may lead to false positives, if the rule is so generic that it matches parts of the code that should not be transformed. In a final step, Spinfer validates the complete generated semantic patch on the complete example files, which may contain a great deal of code other than that of the functions the rule was learned from. If this validation shows that a rule causes false positives, Spinfer specializes the rule according to the specific code fragments that motivated its construction, resulting in a safer, but potentially less widely applicable, semantic patch rule.

Data-flow inconsistencies. A semantic patch rule needs to be able to construct the *After* code from the information found in the *Before* code, to carry out the intended change. Thus, in a completed semantic patch rule-graph, Spinfer reassigns the metavariables, `X0`, `X1`, etc. (see Figure 2), that were local to each fragment, in a way that is consistent across the proposed semantic patch rule. It may occur that no consistent assignment is possible, and some metavariables may remain in the *After* fragments that are not instantiated by the *Before* fragments.

As an example, suppose we add the fourth example of Figure 1 to our set of fragments. This example has no `data` field initialization, and thus it will cause a split from the others, producing a separate semantic patch rule-graph. This rule-graph will indicate removal of the abstract fragment `init_timer(&X0);` and removal of the subsequent fragment `X0.function = X1;`, and the addition of the fragment `setup_timer(X0,X1,X2);`. `X2`, however, represents `0UL`, which is not represented by any of the metavariables of the removed code. In this situation, Spinfer agglomeratively considers subsets of the examples contributing to the semantic patch rule-graph, to determine whether respecializing the metavariables to the contents of the considered subsets can produce a consistent metavariable assignment. In our case, there is only one associated example, and Spinfer eliminates `X2` entirely, replacing it with `0UL` in the generated semantic patch.

3.5 Rule ordering

Finally Spinfer pretty prints each semantic patch rule-graph into a rule, and then orders the rules to form a single semantic patch. This last step is important because graph splitting can produce rules that subsume other rules or rules that must be executed in a certain order to limit the number of false positives. For instance, for the `init_timer` example, a rule matching the `nmi.c` variant must be executed after all other rules that could have consumed the missing `data` field.

To solve these issues Spinfer first looks for semantic patch rules that subsume other rules. Let R_1 and R_2 be two rules and let TP_1 and TP_2 be the sets of true positives and FP_1 and FP_2 the sets of false positives produced by applying R_1 and R_2 respectively, by using Coccinelle on the provided examples. R_1 is said to be subsumed by R_2 if TP_1 is a subset of TP_2 and FP_1 is a superset of FP_2. When Spinfer detects a rule that is subsumed by others it is eliminated from the final semantic patch. Finally, Spinfer orders the remaining semantic patch rules, by looking at the F2 score (a combination of precision and recall that favors high recall) of tentative semantic patches each consisting of a pair of rules. From these results, it does a topological sort to order the complete set of semantic patch rules into a single semantic patch.

4 Evaluation

In this section, we evaluate Spinfer on two datasets composed of real changes from the Linux kernel. We then illustrate a failure case of Spinfer in which it generates an incorrect semantic patch and explain why it was generated this way. With this example, we illustrate how a developer can easily fix a semantic patch to produce a correct one.

4.1 Methodology

Our first dataset is a collection of 40 sets of changes in the Linux kernel, that have been selected to challenge Spinfer, by focusing on the higher levels of the taxonomy. Consequently, it is not intended to be representative of the real taxonomy distribution of Linux kernel changes. The sets include some changes that have been performed using Coccinelle as well as recurring changes found in recent Linux kernel versions v4.16-v4.19, identified using the tool `patchparse` [20]. We refer to this dataset as the *challenging dataset*.

Our second dataset is composed of 40 sets of changes randomly picked from changes to the Linux kernel in 2018. To build this dataset we first identified 175 large-scale changes (changes from one developer affecting at least 10 files) from the changes performed in 2018 and then we randomly selected 40. We refer to this dataset as the *2018 dataset*.

For each dataset we describe the interesting aspects of its taxonomy and we perform two experiments:

The *synthesis experiment* learns semantic patches from the full dataset containing all the files and evaluates the resulting semantic patch on the same set of files. This experiment evaluates the degree to which Spinfer is able to capture the changes found in the examples provided to it. It is relevant to the user who wants to understand a previously performed change. Without Spinfer, such a user has to read through the entire patch and collect all of the different kinds of changes performed, with no way to validate his understanding. Spinfer does both the inference and validation automatically.

The *transformation experiment*, on the other hand, learns semantic patches from a reduced dataset composed of the first 10 changed files, as indicated by the commit author date (ties between files in the same commit are broken randomly), or half of the full dataset if the full dataset contains fewer than 20 files. This experiment evaluates the resulting semantic patch on the portion of the dataset that does *not* include the set of files from which the rules were learned. This experiment is relevant to the user who wants to change new code by bootstrapping a semantic patch.

We recall that Spinfer targets the whole taxonomy, except in terms of control flow where it targets only C_0 and C_1, but not the higher levels. Since not all examples are in Spinfer's targets in terms of the taxonomy, for each experiment we separate the analysis of the results according to whether the examples are in Spinfer's scope or not.

We use classic metrics that are applied for evaluation of program transformation tools: *precision*, *i.e.*, the percentage of changes obtained by applying the inferred semantic patch that are identical to the expected changes in the examples, and *recall*, *i.e.*, the percentage of expected changes in the examples that are obtained by applying the inferred semantic patch. To address the issue of *noise* we compare the changes performed by the semantic patch generated by Spinfer against those performed by a human-written semantic patch created by a Coccinelle expert. In this way, we benefit from the intuition of the human expert to filter out noise. As there are many ways to write a semantic patch for the same change, we do not directly compare the syntax of the generated semantic patch against the human-written one, and instead focus on the results of applying the semantic patch itself.

4.2 Experiments on the *challenging dataset*

The taxonomy of the changes for the *challenging dataset* is shown in Figure 6. The majority of transformations in this dataset require taking into account both control-flow and data-flow relationships, with 21 examples requiring at least control-flow dependencies on unchanged terms (C_2), which is outside of the scope of Spinfer (C_0 or C_1). Three quarters of the transformations have multiple variants including one quarter with variants that need to be performed in a specific order, features that are targeted by Spinfer.

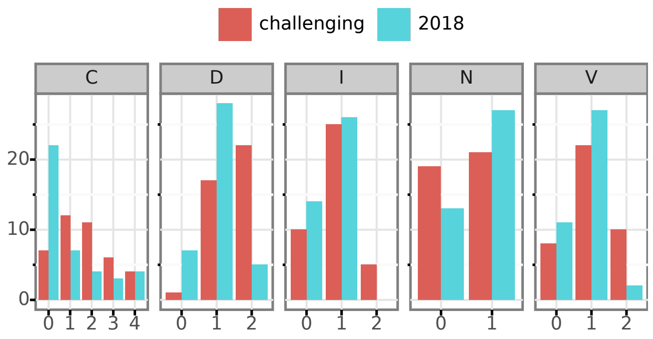


Figure 6: Taxonomy of the two datasets

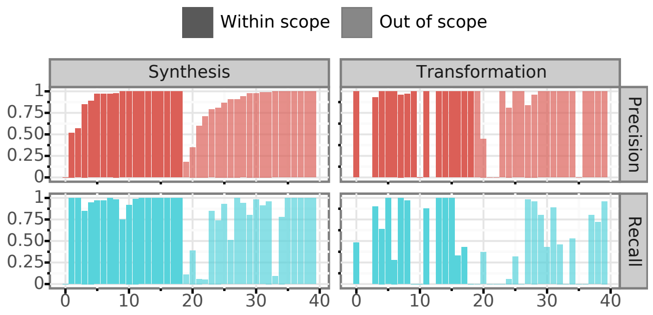


Figure 7: CFG-nodes metrics for the *challenging dataset*

To evaluate Spinfer in terms of the correctness of the transformation. We look at the ratio of correctly modified CFG-nodes, either successfully deleted, or added to the correct location in the CFG of the example. As Spinfer prints around one line of semantic patch per modified node, this roughly translates to the ratio of correct lines in the generated semantic patch. Figure 7 gives the recall and precision for the node modifications produced by the generated semantic patches, with each bar corresponding to one set of changes. The lighter color of the bars on the right indicates that the examples are not in Spinfer target in terms of the taxonomy. Missing values for the transformation experiment at indices 1, 2 and 12 correspond to sets of changes with 2 modified files or fewer. In this case, performing the transformation experiment is not possible as Spinfer needs to learn from at least two files. Other missing values indicate that Spinfer either did not generate anything, or generated a semantic patch that was not applicable to the evaluation dataset. These cases count as having both a recall and a precision of 0.

On average, the semantic patches generated by Spinfer achieve 87% precision and 81% recall for the synthesis experiment, and 86% precision and 49% recall for the transformation experiment. For the part of the examples in Spinfer's scope in terms of the taxonomy, Spinfer obtained 88% precision and 91% recall for the synthesis experiment, and 93% precision and 62% recall for the transformation experiment. Precision and recall are much lower for examples outside Spinfer's scope, averaging 86% precision and 72% recall in

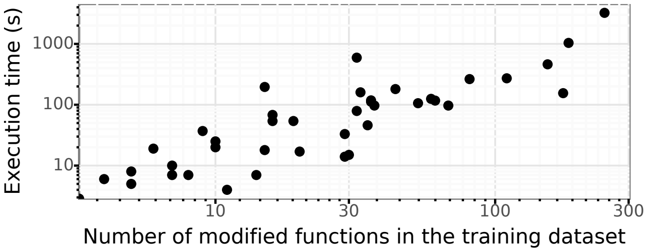


Figure 8: Execution time for the *challenging dataset*

synthesis and 81% precision and 39% recall in transformation.

These results suggest that, for the subset of examples in its scope, Spinfer can be used to infer semantic patches that can perform most of the needed transformations. Also, by carefully examining the produced semantic patches, we believe that most of the non-exhaustive semantic patches in this subset can be modified in a couple of minutes to obtain complete ones. For examples outside Spinfer scope, Spinfer is still able to perform some of the transformations without producing too many false positives.

As shown in Figure 8, Spinfer's execution time is almost linear in the number of modified functions in the training dataset. Spinfer takes less than 50 seconds to infer a semantic patch for examples with 10 or fewer modified functions. We believe that Spinfer's execution time is quite reasonable to be used as a tool to suggest a semantic patch.

4.3 Experiments on the *2018 dataset*

Figure 6 also shows the taxonomy of the changes for the *2018 dataset*. Contrary to the *challenging dataset*, the largest part of the changes in the *2018 dataset* do not require taking complex control flow into account (C_0 and C_1) and thus fit in Spinfer's scope. The kinds of control-flow constraints observed are from one statement to the next, and from one statement to some later statement that is not directly contiguous. Only a few changes require taking into account negative information (C_3), such as verification that a variable is not used in its scope before removing it, or interprocedural information (C_4). However, multiple variants (I_1) and noise (N_1) are very common.

We evaluate the results in the same way as for the *challenging dataset*. Figure 9 gives the recall and precision for the generated semantic patch. For the four C_4 cases in the taxonomy the results are evaluated against the developers' changes directly, rather than against human-written semantic patches, as Coccinelle does not support interprocedural control-flow constraints.

For the *2018 dataset*, the generated semantic patches achieve 85% precision and 83% recall in synthesis, and 87% precision and 62% recall in transformation. Looking only at the examples in the scope of Spinfer, the generated semantic patches obtained 92% precision and 88% recall in synthesis, and 94% precision and 65% recall in transformations, which is very close to the results for the *challenging dataset*. As most of the transformations in our randomly sampled dataset are in Spinfer scope, these results suggest that Spinfer is adapted to real kernel transformation situations, and can generate high quality semantic patches that can be used by kernel developers to perform synthesis or transformation tasks.

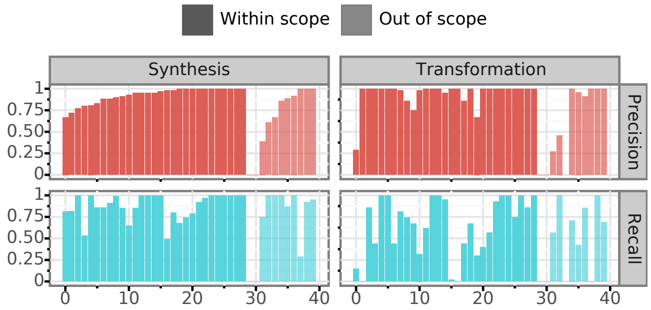


Figure 9: CFG-nodes metrics for the *2018 dataset*

4.4 Analysis of a failure case

We now analyze a typical failure case that gives an intuition for why some examples have a lower recall in transformation than in synthesis.

A case where Spinfer does not generate a high quality semantic patch is for the elimination of the **cpufreq_frequency_table_target** function. For this transformation our human expert wrote the following semantic patch:

```
@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-        CPUFREQ_RELATION_H)
+ cpufreq_table_find_index_h(policy, old_freq)

@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-        CPUFREQ_RELATION_L)
+ cpufreq_table_find_index_l(policy, old_freq)

@@ expression policy, old_freq; @@
- cpufreq_frequency_table_target(policy, old_freq,
-        CPUFREQ_RELATION_C)
+ cpufreq_table_find_index_c(policy, old_freq)
```

The semantic patch composed of three rules replaces the uses of **cpufreq_frequency_table_target** by specialized **cpufreq_table_find_index** functions. The choice of replacement function depends on the constant used as the third argument of **cpufreq_frequency_table_target**.

For this transformation we launched Spinfer on a learning set of only two files and, in under two seconds, Spinfer generated the following semantic patch composed of two rules:

```
@@ expression E0, E1, E2; @@
- E0 = cpufreq_frequency_table_target(E1, E2,
-        CPUFREQ_RELATION_H);
+ E0 = cpufreq_table_find_index_h(E1, E2);
```

```
@@ expression E0, E1; @@
- E0 = cpufreq_frequency_table_target(&E1, E1.cur,
-        CPUFREQ_RELATION_C);
+ E0 = cpufreq_table_find_index_c(&E1, E1.cur);
```

The generated semantic patch does not cover all the cases, and thus it is incomplete. It illustrates two typical reasons for the failure Spinfer: **variant bias** and **over-specialization**.

Variant bias happens when only a subset of variants is present in the learning set. In this case the learning set contains only instances with **CPUFREQ_RELATION_H** and **CPUFREQ_RELATION_C** constants. Consequently Spinfer did not see the third constant and cannot generate a rule for it. Given this semantic patch, however, a developer can easily find the missing constant and complete the semantic patch with copy-pasting and small edits.

Over-specialization is present here in two forms: all rules are assignments and the second rule contains a mandatory field **cur** for the second parameter of both functions. Both of these constraints are incorrect, but they are generated because they are present in all examples in the learning set and because Spinfer prefers precision over recall. Once again, fixing these issues can be very quick for a Linux kernel developer, once the basic outline of the semantic patch is provided.

We have illustrated two typical reasons for the failure of Spinfer, in which our tool generates an incorrect semantic patch for the testing set. These cases happen because Spinfer somewhat overfits the learning set to prevent false positives. However these kinds of failure can be easily and quickly fixed by developers, by using their knowledge of the kernel or by providing more examples to our tool.

5 Related Work

The most closely related works are Sydit [17], Lase [18], AppEvolve [7], Refazer [23], Meditor [30], A4 [11], and GenPat [8], that were already presented in Section 2.

A novelty of Spinfer is its focus on control-flow graphs. CBCD [14] relies on Program Dependence Graphs (PDGs) to find other instances of a known bug in a code base. CBCD is limited to only one example bug, does not produce a script for matching buggy code, and does not address how to fix bugs. By considering commonalities among multiple examples, Spinfer can learn more general rules. By providing a transformation script, Spinfer makes the result understandable to the user and even allows the user to improve the script or adapt it to other uses. Finally, Spinfer-inferred semantic patches both find code needing transformation and describe how to transform it automatically.

Spinfer requires the existence of patches that illustrate a desired change. When the developer does not know how to make a change, he can search for examples in the commit history. Patchparse [20] finds common changes in commit histories of C code. SysEdMiner [19] finds such changes for Java code. Prequel allows the developer to search for commits that match patterns of removed and added code, making it possible to find examples for specific changes [13]. The output from those tools may be used with Spinfer.

6 Conclusion

This paper proposes an approach to automatically inferring Coccinelle semantic patches from a collection of change examples written in C. Our approach considers similar code fragments and control flows among the changes to identify change patterns and construct transformation rules. Generated semantic patches are easily understandable by developers and can be used both for understanding past changes and to perform large scale transformation, with little to no modifications. We have implemented our approach as a tool named Spinfer, and evaluated it on two sets of 40 real changes from the Linux kernel. Our evaluation shown that Spinfer is capable of handling the majority of the real Linux kernel transformation situations by generating semantic patches with high recall and precision in only a few minutes.

We have also identified a taxonomy of challenges that transformation-rule inference tools for systems code must solve, based on the complexity of the control and data flow, and the number of change instances and variants. Such a taxonomy provides level ground for comparing the capabilities of different transformation-rule inference tools, and Spinfer achieves much safer and more comprehensive results than previous tools with respect to the properties defined in the taxonomy. The taxonomy can also be used to guide the future development of Spinfer and other transformation-rule inference tools. In particular, the next frontier for transformation rule inference is to effectively take into account unmodified terms into control-flow dependencies. Typically, for a large-scale change, each changed function contains many times more unmodified lines than changed lines, and the unmodified lines exhibit much greater variety. A challenge is to identify which of the unmodified lines, if any, are relevant to controlling when a change should be performed.

Acknowledgements

This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003) and the ANR ITrans project. We would like to thank the anonymous reviewers and our shepherd, Baris Kasikci, for their feedback.

Availability

Spinfer source code is publicly available at
https://gitlab.inria.fr/spinfer/spinfer.

References

[1] Jesper Andersen and Julia L. Lawall. Generic patch inference. In *ASE*, pages 337–346, 2008.

[2] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *ASE*, pages 382–385, 2012.

[3] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

[4] Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*. Citeseer, 2009.

[5] Kees Cook. treewide: init_timer() -> setup_timer(), October 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b9eaf1872222.

[6] Deepa Dinamani. vfs: change inode times to use struct timespec64, May 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=95582b008388.

[7] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, page 12, 2019.

[8] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. Inferring program transformations from singular examples via big code. In *ASE*, 2019. To appear.

[9] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *ICSE*, pages 351–360, 2011.

[10] Michael Kerrisk. Ks2012: Kernel build/boot testing, September 2012. https://lwn.net/Articles/514278/.

[11] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. A4: Automatically Assisting Android API Migrations Using Code Examples. *arXiv:1812.04894 [cs]*, December 2018. arXiv: 1812.04894.

[12] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, pages 601–614, 2018.

[13] Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *USENIX ATC*, pages 15–26, 2017.

[14] Jingyue Li and Michael D. Ernst. CBCD: cloned buggy code detector. In *ICSE*, pages 310–320, 2012.

[15] Edward S. Lowry and Cleburne W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[16] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With GNU Diff and Patch*. Network Theory Ltd, January 2003.

[17] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[18] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511, 2013.

[19] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Mining Software Repositories (MSR)*, pages 248–256, 2017.

[20] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

[21] Gordon D Plotkin. A note on inductive generalization.

[22] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5(1):135—151, 1970.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE*, pages 404–415, 2017.

[24] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *arXiv*, 2018.

[25] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, November 1987.

[26] Claude Sammut and Geoffrey I. Webb, editors. *TF–IDF (Encyclopedia of Machine Learning)*, pages 986–987. Springer US, Boston, MA, 2010.

[27] Wolfram Sang. tree-wide: simplify getting .drvdata, April 2018. https://lkml.org/lkml/2018/4/19/547.

[28] Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v5.4/process/submitting-patches.html.

[29] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automated deprecated-api usage update for android apps: How far are we? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611. IEEE, 2020.

[30] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 335–346, Piscataway, NJ, USA, 2019. IEEE Press.

[31] Ying Zhao and George Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 515–524. ACM, 2002.

FuZZan: Efficient Sanitizer Metadata Design for Fuzzing

Yuseok Jeon
Purdue University

Wookhyun Han
KAIST

Nathan Burow
Purdue University

Mathias Payer
EPFL

Abstract

Fuzzing is one of the most popular and effective techniques for finding software bugs. To detect triggered bugs, fuzzers leverage a variety of *sanitizers* in practice. Unfortunately, sanitizers target long running experiments—e.g., developer test suites—not fuzzing, where execution time is highly variable ranging from extremely short to long. Design decisions made for developer test suites introduce high overhead on short lived fuzzing executions, decreasing the fuzzer's throughput and thereby reducing effectiveness.

The root cause of this sanitization overhead is the heavy-weight metadata structure that is optimized for frequent metadata operations over long executions. To address this, we design new metadata structures for sanitizers, and propose FuZZan to *automatically* select the optimal metadata structure without any user configuration. Our new metadata structures have the *same* bug detection capabilities as the ones they replace. We implement and apply these ideas to Address Sanitizer (ASan), which is the most popular sanitizer.

Our evaluation shows that on the Google fuzzer test suite, FuZZan improves fuzzing throughput over ASan by 48% starting with Google's provided seeds (52% when starting with empty seeds on the same applications). Due to this improved throughput, FuZZan discovers 13% more unique paths given the same 24 hours and finds bugs 42% faster. Furthermore, FuZZan catches all bugs ASan does; i.e., we have not traded precision for performance. Our findings show that sanitizer performance overhead is avoidable when metadata structures are designed for fuzzing, and that the performance difference will have a meaningful difference in squashing software bugs.

1 Introduction

Fuzzing [33] is a powerful and widely used software security testing technique that uses randomly generated inputs to find bugs. Fuzzing has seen near ubiquitous adoption in industry, and has discovered countless bugs. For example, the state-of-the-art fuzzer American Fuzzy Lop (AFL) has discovered hundreds of bugs in widely-used software [57], while Google has found 16,000 bugs in Chrome and 11,000 bugs in over 160 other open source projects using fuzzing [10]. On its own, fuzzing only discovers a subset of all triggered bugs, e.g., failed assertions or memory errors causing segmentation faults. Bugs that silently corrupt the program's memory state, without causing a crash, are missed. To detect such bugs, fuzzers must be paired with sanitizers that enforce security policies at runtime by turning a silent corruption into a crash. To date, around 34 sanitizers [47] have been prototyped. So far, only the LLVM-based sanitizers ASan, MSan, LeakSan, UBSan, and TSan have seen wide-spread use. For brevity, we use *sanitizers* to refer to such frequently used sanitizers in the rest of the paper.

Unfortunately, sanitizers are designed for developer-driven software testing rather than fuzzing, and are consequently optimized for minimal per-check cost, not startup/teardown of the metadata structure. Consequently, they are based around a shadow-memory data structure wherein the address space is partitioned, and metadata is encoded into the "shadow" memory at a constant offset from program memory. Optimizing for long executions makes sense in the context of developer-driven software testing, which generally verifies correct behavior on expected input, leading to relatively long test execution times. Fuzzing has a more diverse set of inputs that cause both short (i.e., invalid inputs) and long running executions with billions of executions. For example, the Chrome developers use Address Sanitizer (ASan) for their unit tests and long-running integration tests [39]. However, the underlying design decisions that make ASan a highly performant sanitizer for long running tests result in high performance overhead—up to 6.59×—for short executions, as observed in a fuzzing environment[1]. This high overhead reduces throughput, thereby preventing a fuzzer from finding bugs effectively.

We analyze the source of this overhead across a variety of sanitizers, and attribute the cost to heavy-weight metadata structures employed by these sanitizers. For example, Address Sanitizer maps an additional 20TB of memory for each exe-

[1] The average time for a single execution across the first 500,000 tests for the full Google fuzzer test suite is 0.61ms.

cution, Memory Sanitizer (MSan) 72TB, and Thread Sanitizer (TSan) 97TB on a 64-bit platform. The high setup/teardown cost of heavy-weight metadata structures is amortized over the long execution of programs due to the low per-check cost. In contrast, a fuzzing campaign typically consists of massive amounts of short-lived executions, effectively transforming what is a large one-time cost into a large runtime cost. For example, Table 1 indicates that memory management is the main source of overhead for ASan under fuzzing on the Google fuzz test suite, accounting for 40.16% of the total execution time we observe. Memory management is the key bottleneck for using sanitizers with fuzzers, and has to date gone unaddressed.

Instead, increasing the efficiency and efficacy of fuzzing has received significant research attention on two fronts: (i) mechanisms that reduce the overhead of fuzzers [27, 55, 57]; and (ii) mechanisms that reduce the overhead of sanitization on longer running tests and conflicts between sanitizers [25, 37, 38, 52, 54]. These works address fuzzers and sanitizers in isolation, ignoring the core sanitizer design decision to optimize for long running test cases using a heavy-weight metadata structure that limits sanitizer performance in combination with fuzzers. Consequently, optimization of sanitizer memory management for short execution times remains an open challenge, motivated by the need to design sanitizers that are optimal under fuzz testing.

We present FuZZan, which uses a two-pronged approach to optimize sanitizers for short execution times, as seen under fuzzing: (i) two new light-weight metadata structures that trade significantly reduced startup/teardown costs [2] for moderately higher (or equivalent) per access costs and (ii) a dynamic metadata structure switching technique, which dynamically selects the optimal metadata structure during a fuzzing campaign based on the current execution profile of the program; i.e., how often the metadata is accessed. Each of our proposed metadata structures is optimized for different execution patterns; i.e., they have different costs for creating an entry when an object is allocated versus looking up information in the metadata table. By observing the metadata access and memory usage patterns at runtime, FuZZan dynamically switches to the best metadata structure *without* user interaction, and tunes this configuration throughout the fuzzing campaign.

We apply our ideas to ASan, which is the most widely used sanitizer [43, 44, 47]. ASan focuses on memory safety violations—arguably the most dangerous class of bugs, accounting for 70% of vulnerabilities at Microsoft [34]—and has already detected over 10,000 memory safety violations [9, 12, 50] in various applications (e.g., over 3,000 bugs in Chrome in 3 years [50]) and the Linux kernel (e.g., over 1,000 bugs [12, 51]) by using a customized kernel address sanitizer (KASan). We further apply FuZZan to MSan and MOpt-AFL.

FuZZan improves fuzzing throughput over ASan by 52% when starting with empty seeds and 48% when starting with Google's seed corpus, averaged across all applications in the Google fuzzer test suite [11] as part of our input record/replay fuzzing experiment. Due to this improved throughput, FuZZan discovers 13% more unique paths (with an improvement in throughput of 61% compared to ASan) given the standard 24 hour fuzz testing with widely used real-world software and a provided corpus of starting seeds.

Modes	ASan's init time ms (%)	ASan's logging time ms (%)	Memory mgmt. time ms (%)	# page faults
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967

Table 1: Comparison between native and ASan executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite [11]. Times are shown in milliseconds, and % denotes the ratio to total execution time.

Crucially, FuZZan achieves this without *any* reduction in bug-finding ability. Therefore, FuZZan strictly increases the performance of ASan-enabled fuzzing, resulting in finding the *same* bugs in *less* time than using ASan with the same fuzzer.

Our contributions are:

1. Identifying and analyzing the primary source of overhead when sanitizers are used with fuzzing, and pinpointing the sanitizer design decisions that cause the overhead;

2. Designing and implementing a sanitizer optimization (FuZZan) and applying it to ASan; that is, we design several new metadata structures along with a dynamic metadata structure switching to choose the optimal structure at runtime. We also validate the generality of our design by further applying it to MSan and MOpt-AFL;

3. Evaluating FuZZan on the Google fuzzer test suite and other widely used real-world software and showing that FuZZan effectively improves fuzzing throughput (and therefore discovers more unique bugs or paths given the same amount of time).

2 Background and Analysis

We present an overview of fuzzing overhead and ASan (our target sanitizer). Further, we detail the design conflicts between ASan and fuzzing when used in combination.

2.1 Fuzzing overhead

Given the same input generation capabilities, a fuzzer's throughput (executions per second) is critical to its effectiveness in finding bugs. Greater throughput results in more code

[2] Compared to ASan, our min-shadow memory mode reduces the time that startup/teardown functions spend in the kernel by 62% on the first 500,000 tests across the full Google fuzzer test suite.

and data paths being explored, and thus potentially triggers more bugs. Running a fuzzer imposes some overhead on the program, a major component of which is the repeated execution of the target program's initialization routines. These routines—including program loading, `execve`, and initialization—do not change across test cases, and hence result in repeated and unnecessary startup costs. To reduce this overhead, many fuzzers leverage a *fork server*. A fork server loads and executes the target program to a fully-initialized state, and then clones this process to execute each test case. This ensures that the execution of each test case begins from an initialized state, and removes the overhead associated with the initial startup.

Another technique for reducing process initialization costs is *in-process fuzzing*, such as AFL's persistent mode and libFuzzer. In-process fuzzing wraps each test in one iteration of a loop in one process, thus avoiding starting a new process for each test. However, in-process fuzzing generally requires manual analysis and code changes [13, 58]. Additionally, in-process fuzzing requires the target code to be stateless across executions as all tests share one process environment, otherwise the execution of one test may affect subsequent ones, potentially leading to false positives. Consequently, testers should avoid in-process fuzzing for library code using global variables. Bugs found from in-process fuzzing may not be reproducible as it is not always possible to construct a valid calling context to trigger detected bugs in the target function, and side-effects across multiple function calls may not be captured [32]. Because of these limitations, in-process fuzzing is used on stateless functions in libraries, while the fork server model (i.e., out-of-process fuzzing) remains the most general fuzzing mode for fuzzing programs.

2.2 Address Sanitizer

All sanitizers leverage a customized metadata structure [47]. Out of many different metadata schemes, shadow memory (both direct-mapped or multi-level shadow) is the most widely used [4, 14–16, 29, 30, 42, 45, 48, 49, 56]. ASan enforces memory safety by encoding the accessibility of each byte in shadow memory. Allocated (and therefore accessible) areas are marked and padded with inaccessible red zones. In particular, *direct-mapped shadow memory* encodes the validity of the entire virtual memory space, with every 8-bytes of memory mapping to 1-byte in shadow memory. Shadow memory encodes the state of application memory. The 8-bit value k encodes that the 8-k bytes of the mapped memory are accessible. The corresponding shadow memory address for a byte of memory is at:

$$addr_{shadow} = (addr >> 3) + offset$$

where *addr* is the accessed address. Generally, ASan only inserts redzones to the high address side of each object as the preceding object's redzone suffices for the low address side. ASan also instruments each runtime memory access to check if the accessed memory is in a red zone, and if so faults. ASan's effectiveness in detecting hard-to-catch memory bugs has led to its widespread adoption. It has become best practice [47] to use ASan (or KASan [20], the kernel equivalent) with a fuzzer to improve the bug detection capability.

2.3 Overhead Analysis of Fuzzing with ASan

To understand ASan's overhead with fuzzing, we analyze the Linux kernel functions used during fuzzing campaigns. Table 1 shows the overhead added by ASan, broken out across ASan's logging, ASan's initialization, and memory management. Our experiments measure the ratio of the time spent in the kernel functions compared to the total execution time for a number of target programs.

Note that memory management makes up 40.16% of ASan's total execution time, as opposed to 11.49% for the base case, and that memory management is more than double the overhead of ASan's logging and initialization *combined*. ASan's heavy use of the virtual address space results in 4.66× page faults compared to native execution. Our memory management overhead numbers reflect the time spent by the kernel in the four core page table management functions: (i) `unmap_vmas` (24.6%), (ii) `free_pgtable` (4.7%), (iii) `do_wp_page` (8.2%), and (iv) `sys_mmap` (2.6%).

Notably, `unmap_vmas` and `free_pgtable` correspond to 73% of ASan's measured memory management overhead across the four core page table management functions. The execution time for these two functions (`unmap_vmas` and `free_pgtable`) is 10x higher than when executing without ASan. To break this overhead down, when executing a test under the fork server mode, a fuzzer needs to create a new process for each test. During initialization, ASan reserves memory space (20TB total, including 16TB of shadow memory, and a separate 4TB for the heap on 64-bit platforms) and then poisons the shadow memory for globals and the heap. Accessing these pages incurs additional page faults, and thus page table management overhead in the kernel. Note that the large heap area causes sparse page table entries (PTEs), which increase the number of pages used for the page table and memory management overhead.

Existing techniques to deal efficiently with large allocations do not help here. Lazy page allocation of the large virtual memory area used by ASan does not mitigate memory management overhead in this case, as many of the pages are accessed when shadow memory is poisoned. Poisoning forces a copy even for copy-on-write pages, and thus increases page table management cost. During execution, memory allocations and accesses cause additional shadow memory pages to be used, again with page faults and page table management. When the process exits, the kernel clears all page table entries through `unmap_vmas` and releases memory for the page table (via `free_pgtables`). The cost of these two functions are correlated with the number of physical pages used by the process. As fuzzing leads to repeated, short executions, such bookkeeping introduces considerable memory management overhead. In contrast to these active memory management functions, `sys_mmap` only accounts for 7% memory management overhead of ASan. This is the expense for reserving all virtual memory areas. However, large areas that are actively accessed by ASan incur considerable additional expenses as detailed above.

For completeness, we note that our analysis finds that ASan performs excessive "always-on" logging (18.86%) by default, and that ASan's initial poisoning of global variables (10.58%) is inefficient. Combined, these additional sources of overhead account for 29.44% overhead. We address these engineering shortcomings in our evaluation, but they are neither our core contributions nor the choke point in fuzzing with ASan.

3 FuZZan design

FuZZan has two design goals: (1) define new light-weight metadata structures, and (2) automatically switch between metadata structures depending on the runtime execution profile. In this section, we present how we design each component of FuZZan to achieve both goals, as illustrated in Figure 1.

3.1 FuZZan Metadata Structures

To minimize startup/teardown costs while maintaining reasonable access costs, FuZZan introduces two new metadata structures: (i) a Red Black tree (RB-tree) metadata structure, which has low startup and teardown costs, but has high per-access costs; and (ii) min-shadow memory, which has medium startup/teardown costs and low per-access costs (on par with ASan). Table 2 shows a qualitative comparison of the different metadata schemes that we propose in this section, see Table 4 for quantitative results. The RB-tree is optimal for short executions with few metadata accesses as it emphasizes low startup and teardown costs, while min-shadow memory is best suited for executions with a mid-to-high number of metadata accesses as it has lower per metadata access costs while still avoiding the full startup/teardown overhead imposed by ASan's shadow memory.

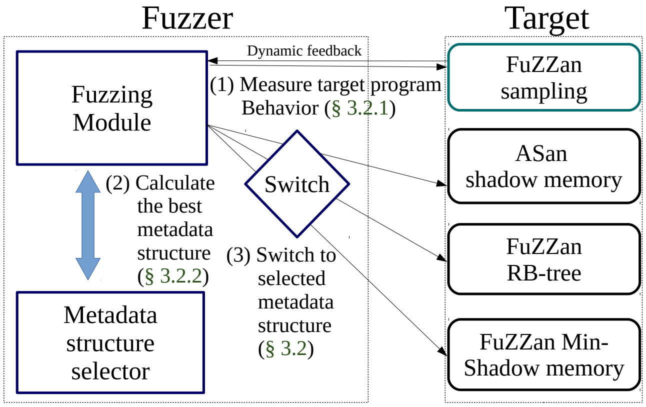

Fuzzer
Target
Dynamic feedback
Fuzzing Module
(1) Measure target program Behavior (§ 3.2.1)
FuZZan sampling
ASan shadow memory
Switch
(2) Calculate the best metadata structure (§ 3.2.2)
(3) Switch to selected metadata structure (§ 3.2)
FuZZan RB-tree
Metadata structure selector
FuZZan Min-Shadow memory

Figure 1: Overview of FuZZan's architecture and workflow.

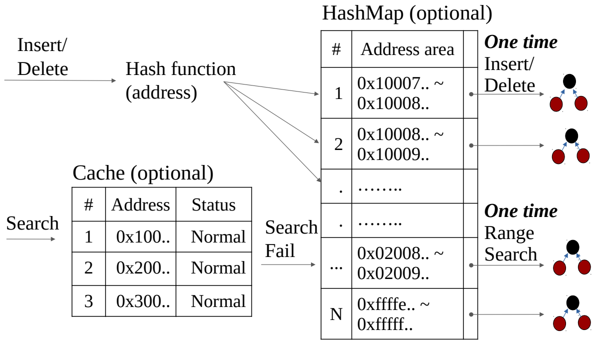
Metadata Structures		Startup/ Teardown Cost	Access Cost
ASan shadow memory		High	Low
FuZZan	Customized RB-tree	Low	High
	Min-shadow memory	Medium	Low

Table 2: Comparison of metadata structures.

3.1.1 Customized RB-Tree

To optimize ASan's metadata structure for test cases where a fuzz testing application only executes for a very short time with few metadata accesses, we introduce a customized RB-tree, shown in Figure 2. Nodes in the RB-tree store the redzone for each object. Although each metadata access operation (insert, delete, and search) in the RB-tree is slower than its counterpart in the shadow memory metadata structure, our RB-tree has the following benefits: (i) low total memory overhead (leading to low startup/teardown overhead); (ii) removal of poisoning/unpoisoning page faults (as each RB-tree node compactly stores the redzone addresses and these nodes are grouped together in memory); and (iii) a faster range search than shadow memory for operations such as `memcpy`. For example, in order to check `memcpy`, ASan must validate each byte individually using shadow memory. However, in our approach, we can verify such operations through only two range queries for `memcpy`'s source and destination memory address range.

In our RB-tree design, when an object is allocated (e.g., through `malloc`), the range of the object's high address redzone is stored in a node of the RB-tree. During a query, if the address range of the target is lower than the start address of the node, we search the left subtree (and vice versa). If the address is not found in the tree, it is a safe memory access. During redzone removal, the requested address range may only be a subset of an existing node's range (and not the full range of a target node in the RB-tree). In this case, the RB-tree



#	Address area
1	0x10007.. ~ 0x10008..
2	0x10008.. ~ 0x10009..
.	……..
.	……..
...	0x02008.. ~ 0x02009..
N	0xffffe.. ~ 0xfffff..

#	Address	Status
1	0x100..	Normal
2	0x200..	Normal
3	0x300..	Normal

Figure 2: Design of FuZZan's customized RB-tree.

deletes the existing RB-tree node, creates new RB-tree nodes which have non-overlapping address ranges (e.g., the left and right side of an overlapped area), and inserts these nodes into the RB-tree. Since we reuse ASan's memory allocator and memory layout (e.g., redzones between objects and a quarantine zone for freed objects), FuZZan provides the same detection capability as ASan.

3.1.2 Min-shadow memory

The idea behind Min-shadow memory (for executions with a mid-to-high number of metadata accesses) is to limit the accessible virtual address space, effectively shrinking the size of the required shadow memory. As the size of shadow memory is a key driver of overhead in the fuzzing environment, this enhances performance.

Figure 3 illustrates how min-shadow memory converts a 64-bit program running in a 48-bit address space to run in a 32-bit address space window (1GB for the stack, 1GB for the heap, and 2GB for the BSS, data, and text sections combined). Note that pointers remain 64 bits wide and the code remains unchanged: the mapped address space is simply restricted, allowing min-shadow memory to have a partial shadow memory map. To shrink a program's memory space, we move the heap (by modifying ASan's heap allocator) and remap the stack to a new address space. Min-shadow memory remaps parts of the address space but programs remain 64-bit programs. To accommodate larger heap sizes, we create additional min-shadow memory binaries with heap sizes of 4GB, 8GB, and 16GB.

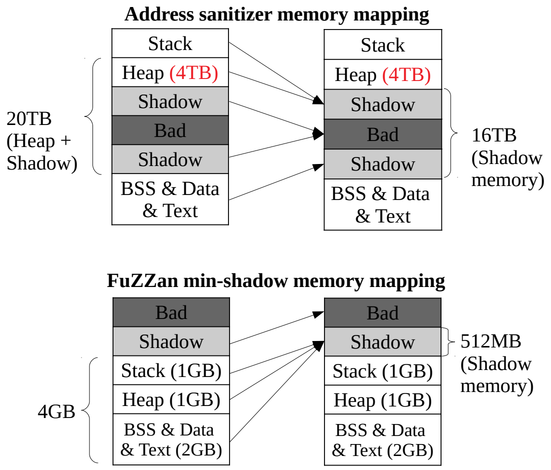


Figure 3: ASan and min-shadow memory modes' memory mapping on 64-bit platforms. ASan (top) reserves 20TB memory space for heap and shadow memory, conversely, min-shadow memory mode (bottom) reserves 4512MB memory space for heap and shadow memory. Each application's stack, heap, and other sections (BSS, data, and text) map to the corresponding shadow regions. Further, the shadow memory region is mapped inaccessible.

Our approach allows testing 64-bit code with 64-bit pointers without having to map shadow tables for the entire address space. We disagree with the recommendation of the ASan developers to compile programs as 32-bit executables, as changing the target architecture, pointer length, and data type sizes will hide bugs. Furthermore, min-shadow memory provides greater flexibility compared to using the `x32` ABI [53] mode (i.e., running the processor in 64-bit mode but using 32-bit pointers and arithmetic, limiting the program to a virtual address space of 4GB), as min-shadow memory can provide various heap size options.

3.2 Dynamic metadata structure switching

Dynamic metadata structure switching automatically selects the optimal metadata scheme based on observed behavior. At the beginning of a fuzzing campaign, dynamic metadata structure switching assesses the initial behavior and then periodically samples behavior, adjusting the metadata structure if necessary. Our intuition for dynamic metadata structure switching is that, during fuzzing, metadata access patterns and memory usage remain similar across runs and change in phases. While the fuzzer is mutating a specific input, the executions of the newly created inputs are *similar regarding their control flow and memory access patterns* compared to the source input. However, new coverage may lead to different execution behaviors. We therefore design a *dynamic metadata structure switching* technique that periodically and conditionally samples the execution and adjusts the underlying metadata structure according to the observed execution behavior.

Dynamic metadata structure switching compiles the program in four different ways in preparation for fuzzing: ASan, RB-tree, min-shadow memory, and sampling mode. The sampling mode repeatedly samples the runtime parameters and then selects the optimal metadata structure. The selection of the optimal metadata structure is governed by FuZZan's metadata structure switching policy.

3.2.1 Sampling mode

The sampling mode measures the behavior of the target program using the min-shadow memory-1GB metadata mode and, based on the behavior, reports the currently optimal metadata structure. The sampling mode profiles the following parameters: (i) the number of metadata accesses during insert, delete, and search; and (ii) memory consumption. Note that this information can be collected by simple counters: profiling is therefore light-weight.

Dynamic metadata structure switching starts in sampling mode and selects the optimal mode based on the observed behavior. Dynamic metadata structure switching then periodically (e.g., every 1,000 executions) and conditionally (e.g., when the fuzzer starts mutating a new test case) samples executions to select the optimal metadata structure based on the current behavior. To reduce the cost of periodic sampling, dynamic metadata structure switching implements a continuous back-off strategy that gradually increases the sampling interval as long as the metadata structure does not change (similar to TCP's slow-start [17]). Note that bugs may be triggered during sampling mode. As such, we maintain ASan's error detection capabilities while sampling to ensure that we do not miss any bugs.

3.2.2 Metadata structure switching policies

Our metadata structure switching policy is based on a mapping of metadata access frequency to the corresponding metadata structure. This heuristic is relatively simple in order to achieve a low sampling overhead. To determine the best cutoff points, we compile all 26 applications in Google's fuzzer test suite in two different ways: RB-tree and min-shadow memory. We then test these different configurations against 50,000 recorded inputs and determine the best metadata structure depending on the observed parameters, measuring execution time. Profiling reveals that the frequency of metadata access (insert, delete, and search) is the primary factor that influences metadata structure overhead, which confirms our original assumption. In this policy, depending on the metadata access frequency, we select different metadata structures (based on statistics from profiling): RB-tree if there are fewer than 1,000 accesses; and min-shadow memory if there are more than 1,000 accesses. Additionally, if the selected heap size goes beyond a threshold, we sequentially switch to other modes (min-shadow memory-4G, 8G, 16G, and ASan), thus increasing heap memory for continuous fuzzing.

4 Implementation

We implement FuZZan's two metadata structures and dynamic metadata structure switching mode on top of ASan in LLVM [28] (version 7.0.0). We support and interact with AFL [57] (version 2.52b). To address the other sources of overhead in ASan (shown in Table 1), we also implement two additional optimizations: (i) removal of unnecessary initialization; and (ii) removal of unnecessary logging. Our implementation consists of 3.5k LOC in total (mostly in LLVM, with minor extensions to AFL).

RB-tree. The RB-tree requires modifications to ASan's memory access instrumentation, as our RB-tree is not based on a shadow memory metadata structure. Thus, we modify all memory access checks, including interceptors, to use the appropriate RB-tree operations instead of the equivalent shadow memory operations. As an optimization, and for compatibility with min-shadow memory mode, the RB-tree mode also reserves 1GB for the heap memory allocator. A compact heap reduces memory management overhead. The RB-tree mode is used when fuzz tests only execute for a very short time with few metadata accesses (i.e., they allocate relatively a small amount of memory).

Min-shadow memory. Unlike the RB-tree, we are able to repurpose ASan's existing memory access checks, as the min-shadow memory metadata structure is based on a shadow memory scheme. To shrink a 64-bit program's address space, we modify ASan's internal heap setup and remap the stack using Kroes et al.'s linker/loader tricks [22]. More specifically, based on this script, we hook `__libc_start_main` using "`LD_PRELOAD`" and then `remap` the stack to a new address, update `rbp` and `rsp`, and then call the original `__libc_start_main`. This allows us to reduce ASan's shadow map requirements from 16TB of mapped (but not necessarily allocated) virtual memory to 512MB (1 bit of shadow for each byte in our 4GB address space window). We also create an additional 192MB shadow memory for ASan's secondary allocator and dynamic libraries (which are remapped above the stack). Finally, we implement four different min-shadow memory modes with increasing heap sizes (1GB, 4GB, 8GB, and 16GB) to handle the different memory requirements of a variety of programs.

Heap size triggers. As previously stated, min-shadow memory is configured for different heap sizes. We therefore use out of memory (OOM) errors to trigger callbacks that notify FuZZan to increase the heap size.

AFL modifications. The target program is compiled once per FuZZan mode. By default, AFL uses a random number generator (RNG) to assign an ID to each basic block within the target program. Unfortunately, this would result in the same input producing different coverage maps across the set of compiled targets, breaking AFL's code coverage analysis. We therefore modify AFL to use the same RNG seed across the set of compiled targets. This ensures that the same input produces the same coverage map across all compiled variants.

Removing unnecessary initialization. ASan makes a number of global constructor calls on program startup, performing several `do_wp_page` calls for copy-on-write. These constructor calls are unnecessarily repeated each time AFL executes a new test input, leading to redundant operations. Unfortunately, the AFL fork server is unaware of ASan's initialization routines. Therefore, to remove unnecessary (re-)initialization across fuzzing runs, we modify ASan's LLVM pass so that global variable initialization occurs *before* AFL's fork server starts. This is achieved by adjusting the priority of global constructors which contain ASan's initialization function.

Removing unnecessary logging. ASan provides logging functionality for error reporting (e.g., saving allocation sizes and thread IDs during object allocation). Unfortunately, this logging functionality introduces additional page faults and performance overhead. However, this logging is unnecessary because fuzzing inherently enables replay by storing test inputs that trigger new behavior. Complete logging information can be recovered by replaying a given input with a fully-instrumented program. We therefore identify and disable ASan's logging functionality (e.g., `StackDepot`) for fuzzing runs, allowing it to be reenabled for reportable runs.

5 Evaluation

We provide a security and performance evaluation of FuZZan. First, we verify that FuZZan and ASan have the same error-detection capabilities. Second, we evaluate the efficiency of FuZZan's new metadata structures and dynamic metadata structure switching mode using deterministic input from a record/replay infrastructure to ensure fair comparisons. Next, to consider the random nature of fuzzing and to show FuZZan's real-world impact, we evaluate FuZZan's efficiency without deterministic input. Here we evaluate the number of code paths found by FuZZan in a 24 hour time period, demonstrating the impact of FuZZan's increased performance. We also measure FuZZan's bug finding speed by using known bugs in Google's fuzzer test suite to verify that FuZZan maximizes fuzzing execution speed while providing the exact same bug detection capabilities as ASan. Finally, we port FuZZan to another sanitizer (MSan) [48] and another AFL-based fuzzer (MOpt-AFL) [31] to verify its flexibility.

Evaluation setup. All of our experiments are performed on a desktop running Ubuntu 18.04.3 LTS with a 32-core AMD Ryzen Threadripper 2990WX, 64GB of RAM, 1TB SSD, and Simultaneous MultiThreading (SMT) disabled (to guarantee a single fuzzing instance is assigned to each physical core). Across all experiments, we apply FuZZan to AFL's fork server mode, which is a widely-used and highly optimized out-of-process fuzzing mode. We evaluate FuZZan on all applications in the Google fuzzer test suite [11] and other widely used real-world software.

Evaluation strategy. Evaluating fuzzing effectiveness is challenging. In a recent study of how to evaluate fuzzing by Klees et. al. [21], the authors find that the inherent randomness of the fuzzer's input generation can lead to seemingly large but spurious differences in fuzzing effectiveness. However, we are at an advantage as we do not need to compare different fuzzers nor do we change the input generation. We therefore record the fuzzer-generated inputs during a regular run of AFL, and then replay these recorded inputs to compare our different ASan optimizations to the same baseline, effectively controlling for randomness in input generation by using the same input for all experiments. For our experiments we record the first 500,000 executions for replay, yielding a large enough test corpus for reasonable performance comparisons. We also undertake a real-world fuzzing campaign (i.e., without inhibiting fuzzing randomness by record/replay) to measure FuZZan's real-world impact on code path exploration. Finally, Klees et. al. demonstrate the importance of the initial seed(s) when evaluating fuzz testing, as performance can vary substantially depending on what seed is used. We therefore compare two scenarios: (i) starting with the empty seed; and (ii) starting with a set of valid seeds (we use Google's provided seeds for the input record/replay experiment and randomly selected seeds of the right file type for our real-world fuzz testing).

CWD (ID)	**Good tests**	**Bad tests**
	(Pass/Total)	**(Pass/Total)**
Stack-based Buffer Overflow (121)	2,432 / 2,432	2,314 / 2,432
Heap-based Buffer Overflow (122)	1,594 / 1,594	1,328 / 1,594
Buffer Under-write (124)	682 / 682	641 / 682
Buffer Over-read (126)	524 / 524	359 / 524
Buffer Under-read (127)	682 / 682	641 / 682
Total	5,914 / 5,914	5,283 / 5,914

Table 3: Three different metadata structure modes' detection capability based on the Juliet Test Suite for memory corruption CWEs. FuZZan and ASan have identical results. Good tests have no memory corruption to check for false positives. Bad tests are intentionally buggy to check for false negatives.

5.1 Detection capability

We verify that FuZZan and ASan detect the same set of bugs in three different ways. First, we use the NIST Juliet test suite [35], which is a collection of test cases containing common vulnerabilities based on Common Weakness Enumeration (CWE). We use the full Juliet test suite for memory corruption CWEs to verify FuZZan's capability to detect the same classes of bugs as ASan, without introducing false positives or negatives. Second, to verify that FuZZan and ASan also have the same detection capability under fuzz testing, we use the Google fuzzer test suite and our recorded input corpus. Finally, we leverage the complete set of ASan's public unit tests as a further sanity check.

For the Juliet test suite (Table 3), we select CWEs related to memory corruption bugs and obtain the same detection results from the three different modes (ASan's shadow memory, RB-tree, and min-shadow memory). To validate FuZZan against ASan on the Google fuzzer test suite, we compare AFL crash reports across the full set of target programs in the Google fuzzer test suite with our recorded inputs (to identify both false positives and false negatives). Note that we force ASan to crash (the default setting under fuzz testing) when a memory error happens as fuzzers depend on program crashes to detect bugs. As expected, FuZZan's different modes all obtain the same crash results as ASan. However, we encounter minor differences between FuZZan and ASan when sanity-checking on the ASan unit tests. These differences are due to internal changes we made when developing FuZZan, such as min-shadow memory's changed memory layout (failed test cases include features such as fixed memory addresses).

Modes	Empty seed			Provided seed		
	time (s)	vs. Native (%)	vs. ASan (%)	time (s)	vs. Native (%)	vs. ASan (%)
Native	199	-	-	274	-	-
ASan	809	306	-	1,105	303	-
RB-tree	1,541	673	90	3,308	1,106	199
Min-1G	443	122	-45	632	131	-43
Min-4G	465	133	-43	666	143	-40
Min-8G	467	134	-42	685	150	-38
Min-16G	477	139	-41	710	159	-36

Table 4: Comparison between four min-shadow memory modes, RB-tree, Native, and ASan execution overhead during input record and replay fuzz testing with empty and provided seed sets. The time (s) indicates the average of all 26 applications' execution time during testing. Positive percentage (e.g., 20%) denotes overhead while negative percentage indicates a speedup.

5.2 Efficiency of new metadata structures

We perform input record/replay fuzz testing to evaluate the effectiveness of FuZZan's new metadata structures. Doing so isolates the effects of our metadata structures by removing most of the randomness/variation from a typical fuzzing run.

Over the full Google fuzzer test suite, the RB-tree, without any other optimization, shows shorter execution times than ASan if the target application has less than 1,000 metadata accesses; conversely, the RB-tree is slower than ASan when the target application has more than 1,000 metadata accesses. On average, as shown in Table 4, several applications in the Google fuzzer test suite have more than 1,000 metadata accesses, and so RB-tree is overall slower than ASan on average.

Despite being slower on average, the RB-tree can be faster on individual applications and inputs. For instance, FuZZan in RB-tree mode demonstrates a 19% performance improvement (up to 45% faster) for 15 applications (the remaining 11 applications show higher overhead compared to ASan) when benchmarked using the inputs generated from an empty seed. On the subset of applications for which seeds are provided, RB-tree shows less performance improvement (17% and up to 39% faster) for 14 applications (the remaining 12 applications show higher overhead than ASan) when benchmarked using inputs generated from those seeds as provided seeds help to create valid input, lengthening execution times and thus metadata accesses. Note that RB-tree shows the best fuzzing performance when the target application (e.g., `c-ares`) has less 1,000 metadata access. Additionally, even for applications where RB-tree is slower across all inputs, it is still *faster* on inputs with few metadata accesses. The variable performance of RB-tree, which is highly dependent on the number of metadata accesses, highlights the need for dynamic metadata structure switching to automatically select the optimal metadata structure.

Modes	Empty seed			Provided seed		
	time (s)	vs. Native (%)	vs. ASan (%)	time (s)	vs. Native (%)	vs. ASan (%)
Logging-Opt.	613	208	-24	891	225	-19
Init-Opt.	686	244	-15	987	260	-11
Logging+Init	552	177	-32	826	201	-25
Min-Shadow	443	122	-45	632	131	-43
Min-Shadow-Opt.	385	93	-52	574	109	-48
Dynamic	387	94	-52	578	111	-48

Table 5: Comparison between FuZZan's three different optimization modes, native min-shadow memory (1G) mode, and min-shadow memory (1G) mode with FuZZan's two optimizations, and dynamic metadata structure switching (Dynamic) mode execution overhead during all 26 applications' input record and replay fuzz testing.

Modes	ASan's init time ms (%)	ASan's logging time ms (%)	Memory manage time ms (%)	Page fault #
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967
Min	0.10 (9.51%)	0.01 (1.33%)	0.24 (24.77%)	7,386
Min-Opt.	0.00 (0.00%)	0.00 (0.00%)	0.24 (24.71%)	6,139

Table 6: Comparison between native, ASan, min-shadow memory (1G), two optimizations with min-shadow memory executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite. Times are shown in milliseconds, and % denotes the ratio between single execution time and each section execution's time.

Min-shadow memory mode, without additional optimization, outperforms ASan on all 26 programs (for both empty and provided seeds), as shown in Table 4. More specifically, the average improvement is 45% when starting with an empty seed and 43% when starting with the provided seeds. While different min-shadow memory heap configurations show gradual increases in memory overhead (from 1GB to 16GB, in line with the heap size), all of them outperform ASan (at worst, min-shadow memory is still 36% faster than ASan with a provided seed).

Additionally, both metadata configurations can utilize our two engineering optimizations; i.e., removing logging and modifying ASan's initialization (as described in § 4). Table 5 shows that the average improvement of removing unnecessary logging is 24% when starting with an empty seed and 19% when starting with the provided seeds. Similarly, modifying the initialization sequence improves performance by 15% when starting with an empty seed and by 11% when starting with the provided seeds. Combining the two engineering optimizations with min-shadow memory demonstrates synergistic effects: the combined performance is 52% (7% better than native min-shadow memory) faster for empty seeds, and 48% (5% better than native min-shadow memory) faster for provided seeds.

Overall, FuZZan's metadata structures show better performance than ASan's shadow memory for all 26 Google fuzzer test suite applications. As shown in Table 6, the main reasons for FuZZan's improvement are: (i) the smaller memory space reduces memory management overhead as page table management is more lightweight and incurs fewer page faults, (ii) our two engineering optimizations further reduce overhead and number of page faults by removing unnecessary operations, and (iii) the min-shadow memory mode has the same $O(1)$ time complexity for accessing target shadow memory as accessing the original ASan metadata. However, we also observe that the RB-tree is faster than min-shadow memory for some configurations and programs (e.g., `c-ares-CVE`). This motivates the need for dynamic metadata structure switching, which observes program behavior and dynamically selects the best metadata structure based on this behavior.

5.3 Efficiency of dynamic metadata structure

As described in § 3.2, the dynamic metadata structure switching mode leverages runtime feedback to select the optimal metadata structure, dynamically tuning fuzzing performance according to runtime feedback. The intuition behind the dynamic metadata structure switching mode is that (i) no single metadata structure is best across all applications, (ii) the best metadata structure is not known a priori, so the analyst cannot pre-select the optimal metadata structure, and (iii) fuzzing goes through phases, e.g., alternating between longer running tests (e.g., exploring new coverage) and shorter running tests (e.g., invalid input mutations searching for new code paths). A consequence of the phases of fuzzing is that the same metadata structure is not optimal for every input to a given application. To verify the effectiveness of dynamic metadata structure switching, which is implemented based on these intuitions, we apply dynamic metadata structure switching mode to fuzz testing for seven widely used applications for fuzzing and all 26 applications' in Google's fuzzer test suite.

Our evaluation of dynamic metadata structure switching validates our intuitions, as shown in Figure 4. Observe that different applications are dominated by different metadata structures, e.g., `c-ares` for RB-tree and `pngfix` for min-shadow memory. This is because dynamic metadata structure switching automatically selects the optimal metadata structure (which is unknown a priori). Because dynamic metadata structure switching is automatic, it prevents users from making errors such as selecting RB-tree for applications with a large number of metadata accesses, and removes the need for any user-driven profiling to make metadata decisions. Further, dynamic metadata structure switching scales alongside with the required memory of applications as it increases when the fuzzer finds deeper test cases, as evidenced by `size`, `pngfix`, or `nm` switching to different min-shadow memory modes (4GB, 8GB, and 16GB heap sizes), without user intervention. Without dynamic metadata structure switching, inefficient min-shadow memory modes would be used at the beginning of fuzzing campaigns, or users would have to pause and restart fuzzing campaigns to change metadata modes.

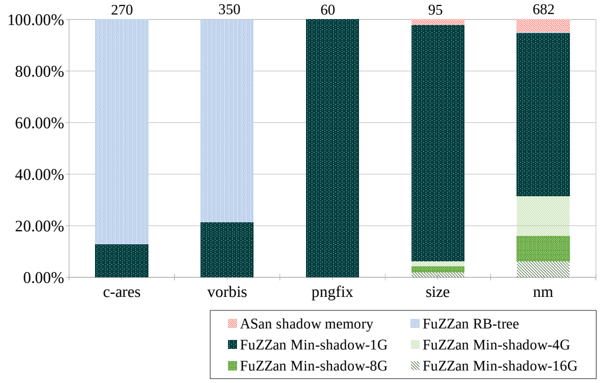


Figure 4: Evaluating the frequency of metadata structure switching and each metadata structure selection over the first 500,000 tests each for c-ares and vorbis in Google's fuzzer test suite and pngfix, size, and nm. The number on each bar indicates the total metadata switches.

As an extreme example highlighting the need for automatic metadata switching, the `nm` benchmark changes metadata structures 682 times, underscoring the infeasibility of having a human analyst determine the single best metadata structure.

As a result of these factors, FuZZan's dynamic metadata structure switching mode improves performance over ASan by 52% when starting with empty seeds and 48% when starting with non-empty seeds. Further, ASan has 306% and FuZZan has 94% (212% less) overhead with empty seeds and ASan has 303% and FuZZan has 111% (192% less) overhead with non-empty seeds compared to native execution. Note that dynamic metadata structure switching has identical fuzzing performance to using min-shadow memory with 1GB heap alone, and improves performance over RB-tree up to 870%. Consequently, automating metadata selection is not adding noticeable overhead, while substantially improving user experience. We recommend using dynamic metadata structure switching mode for the following four reasons: (i) if the target application exceeds FuZZan's heap memory limit (1GB), dynamic metadata structure switching automatically increases the heap size for the few executions that require it (a fixed heap size results in *false positive crashes* due to heap memory exhaustion), (ii) preventing users from selecting an incorrect metadata structure, (iii) using only one metadata structure (e.g., min-shadow memory) may miss the opportunity to further improve throughput, as, in some cases, RB-tree (or some future metadata structure) may be faster than min-shadow memory; (iv) manually selecting a metadata structure requires extra effort (e.g., measuring each metadata structure's efficiency for the target application), which dynamic metadata structure switching mode avoids by automatically selecting the optimal metadata structure.

Programs	Native		ASan		FuZZan	
	exec \#	path \#	exec \#	path \#	exec \# (%)	path \# (%)
cxxfilt	86M	2,769	33M	2,442	51M (55%)	2,651 (9%)
file	29M	1,126	7M	763	9M (29%)	845 (11%)
nm	51M	1,272	7M	822	12M (71%)	872 (6%)
objdump	95M	883	15M	567	17M (13%)	595 (5%)
pngfix	36M	971	18M	912	33M (83%)	982 (8%)
size	52M	703	17M	626	32M (88%)	656 (5%)
tcpdump	70M	3,587	11M	1,540	20M (82%)	2,032 (32%)
Total	419M	11,311	108M	7,672	174M (61%)	8,633 (13%)

Table 7: Evaluating FuZZan's total execution number and unique discovered path for 24 hours fuzz testing with provided seeds. The (M) denotes 1,000,000 (one million) and ratio (%) is the ratio between ASan and FuZZan.

Programs	ASan	FuZZan		Type (source)
	TTE (s)	TTE (s)	rate (%)	
c-ares	45	25	46	BO (ares_create_query.c:196)
json	29	11	61	AF (fuzzer-parse_json.cpp:50)
libxml2	7,314	4,194	43	BO (CVE-2015-8317)
openssl-1.0.1f	443	336	24	BO (t1_lib.c:2586)
pcre2	7,056	4,020	43	BO (pcre2_match.c:5968)
Total	14,887	8,586	42	-

Table 8: Evaluating FuZZan's bug finding speed. The TTE denotes the mean time-to-exposure. The AF is assertion error and the BO denotes buffer overflow.

5.4 Real-world fuzz testing

Our experiments validating FuZZan use a record/replay approach to avoid any impact of randomness, allowing meaningful comparisons to a baseline. However, real-world fuzzing is highly stochastic, and so we also evaluate FuZZan in the context of several real-world end-to-end fuzzing campaigns without deterministic input record/replay. For this experiment, we select the following widely used programs: `cxxfilt`, `nm`, `objdump`, `size` (all from binutil-2.31), `file` (version 5.35), `pngfix` (from libpng 1.6.38) and `tcpdump` (version 4.10.0). Klees et al. [21] select and test `cxxfilt`, `nm`, and `objdump` in their fuzzing evaluation study. The remaining four programs (`size`, `file`, `pngfix`, and `tcpdump`) are widely tested by recent fuzzing works [1, 3, 6, 26, 36, 46]. For each binary, we run a fuzzing campaign. Each campaign is conducted for 24 hours and repeated five times. We measure the number of total executions and discovered unique paths when fuzzing with seeds from the seed corpus of each program with the right type file and three different configurations: native, ASan, and FuZZan's dynamic metadata structure switching mode, and report the mean over the five campaigns.

As a result, FuZZan improves throughput over ASan by 61% (up to 88%). Interestingly, FuZZan discovers 13% more unique paths given the same 24 hours time due to improved throughput. Our evaluation also shows that improved throughput increases the possibility of finding more bugs in the same amount of time, as we discuss next.

Modes	time (s)	vs. Native (%)	vs. MSan (%)	vs. MSan nolock (%)
Native	146	-	-	-
MSan	14,074	9,575	-	-
MSan-nolock	386	165	-97	-
Min-16G	335	130	-98	-13

Table 9: Comparison between Native, MSan, MSan-nolock, and min-shadow memory execution overhead during input record and replay fuzz testing with provided seed sets. MSan-nolock disables lock/unlock for MSan's logging depots. Time (s) indicates the average of execution time. Positive percentages denote overhead, negative percentages denote speedup.

5.5 Bug finding effectiveness

FuZZan increases throughput while maintaining ASan's bug detection capability, potentially enabling it to find more bugs. To demonstrate this, we evaluate FuZZan's bug finding speed and compare it to a fuzzing campaign with ASan. In this evaluation, we target five applications in Google's fuzzer test suite. These applications are chosen because we found bugs in them (using ASan and dynamic metadata structure switching mode) within a 24 hour fuzzing campaign. We use the seeds provided by the test suite and repeated each campaign five times. Note that we do not replay recorded inputs during these campaigns, instead letting the fuzzer generate random inputs. Table 8 shows the mean time (over five campaigns) to find each bug. Notably, FuZZan finds all bugs up to 61% (mean 42%) faster than ASan, and is faster in all cases. This experiment emphasizes our belief that throughput is paramount when fuzzing with sanitizers.

5.6 FuZZan Flexibility

Appling FuZZan to Memory Sanitizer. Like ASan, numerous sanitizers use shadow memory for their metadata structure [47]. For example, other popular sanitizers, such as Memory Sanitizer (MSan) [48] and Thread Sanitizer (TSan) [42], also rely on shadow memory for metadata. FuZZan optimizes sanitizer usage of shadow memory *without* modifying the stored shadow information or how the sanitizer uses that information. Consequently, porting our shadow metadata improvements in FuZZan from ASan to other sanitizers is a simple engineering exercise. To demonstrate this, we port FuZZan to MSan. In so doing, we shrink MSan's memory space to implement min-shadow memory 16G for MSan (1GB for the stack, 16GB for the heap, and 2GB for the BSS, data, and text sections combined). We only implement one metadata mode for our MSan proof-of-concept to validate our claim that applies FuZZan to other shadow memory based sanitizers is an engineering exercise.

Table 9 summarizes MSan's performance overhead on different modes for all 26 evaluated applications. Initially, min-shadow memory shows high overhead—around 96 times native. Analyzing this, we found that MSan's `fork()` interceptor locks all logging depots before `fork()` and similarly unlocks them afterwards to avoid deadlocks. However, as explained in § 4, locking/unlocking logging depots is unnecessary for fuzzing because these logging depots exist for bug reporting and fuzzing inherently enables replay by storing test inputs when the fuzzer finds bugs. We thus disable these lock/unlock functions to create the MSan-nolock mode, which has reasonable overhead (2.6 times that of native).

FuZZan's MSan min-shadow memory 16G mode shows 13% performance improvement compared to MSan-nolock mode, demonstrating FuZZan's efficacy when applied to MSan. We expect that additional optimization and the application of the dynamic switch mode will lead to even higher performance improvement. We leave this engineering as future work.

Applying FuZZan to MOpt-AFL. FuZZan is not coupled to a particular fuzzer or fuzzer version. Most modern fuzzers [2, 3, 31, 31] extend AFL, so our approach applies broadly. To demonstrate this, we apply FuZZan to MOpt-AFL [31], which is an efficient mutation scheduling scheme to achieve better fuzzing efficiency. We modify MOpt-AFL to add FuZZan's profiling feedback and dynamic metadata switching functions. To measure FuZZan's impact on MOpt-AFL, we select seven real-world applications (the same set as Table 7) and fuzz them for 24 hours each, repeating the experiment five times to control for randomness in the results. On average, ASan-MOpt-AFL mode discovers 85% more unique paths given the same 24 hours time due to MOpt-AFL's effectiveness compared to ASan. Notably, FuZZan-MOpt-AFL mode discovers 112% more unique paths (27% higher than ASan-MOpt-AFL) due to the improved throughput.

6 Discussion

In this section, we summarize some potential areas for future work, a possible security extension enabled by FuZZan, and lessons learned in designing FuZZan.

Removing conflicts between sanitizers. ASan's shadow memory scheme conflicts with other sanitizers that are also based on shadow memory, e.g., MSan and TSan. Each sanitizer interprets the shadow memory in a mutually exclusive manner, prohibiting the use of multiple concurrent sanitizers. For example, ASan uses shadow memory as a metadata store, while MSan prohibits access to the same memory range. FuZZan's new metadata structures can be adapted to avoid this conflict, and enable true composition of sanitizers, since we use lightweight, independent metadata structures. Each sanitizer can map its own instance of our metadata structure, and all sanitizers may coexist in a single process. However, some engineering effort is required to port sanitizers to our new metadata structures. An alternate approach would be to have one metadata structure that stores information for all sanitizers. Whether having a unified metadata structure or a metadata structure per sanitizer is more efficient is an interesting research question.

Possible security extension. Unfortunately, ASan's virtual memory requirements directly conflict with fuzzers' abilities to detect certain out-of-memory (OOM) bugs. For example, fuzzers typically limit memory usage to detect OOM errors when parsing malformed input. However, ASan's large virtual memory requirement masks OOM bugs, leaving them undetected because of the difficulty of setting precise memory limits. Consequently, using a compact metadata structure with ASan not only improves performance, but also can enable an extension of ASan's policy to cover OOM bugs.

Lessons Learned. Our initial metadata design leveraged a two-layered shadow memory metadata structure that split metadata lookups into two parts: a lookup into a top-level metadata structure, followed by a lookup into a second-level metadata structure a la page tables. While this design vastly reduced memory consumption and management overhead, the additional runtime cost per metadata access of the additional indirection resulted in the two-layer structure being slower than ASan in all cases.

For dynamic metadata structure switching, we evaluated two additional policies: (i) utilizing more detailed metadata access information such as each object type's (e.g, stack) metadata access (e.g., insert) count and each operation's microbenchmark results, and (ii) running each metadata mode, measuring their execution time, and selecting the fastest metadata mode. In our evaluation, the additional sampling complexity of these policies outweighed any gains from more precisely selecting a metadata structure.

7 Related Work

7.1 Reducing Fuzzing Overhead

Several approaches reduce the overhead of fuzzing. One approach is to reduce the execution time of each iteration. AFL supports a deferred fork server which requires a manual call to the fork server. The analyst is encouraged to use the deferred fork server, and manually initiate the fork server as late as possible to reduce, not only overhead from linking and libc initializations, but also overhead from the initialization of the target program. Deferred mode, however, cannot reduce the teardown overhead of heavy metadata structures. AFL's persistent mode and libFuzzer eliminate the overhead from creating a new process. However, these approaches require manual effort, and users must know the target programs. Xu et al. [55] implement several new OS primitives to improve the efficiency of fuzzing on multicore platforms. Especially, by supporting a new system call, `snapshot` instead of `fork`, they reduce the overhead of creating a process. Moreover, they reduce the overhead from file system contention through a dual file system service.

However, this approach requires kernel modifications for the new primitives, and does not reduce the overhead of sanitizers.

Another approach is to improve fuzzing itself so that it can find more crashes within the same amount of executions. AFLFast [3] adopts a Markov chain model to select a seed. If inputs mutated from a seed explore more new paths, the seed has higher probability to be selected. With given target source locations, AFLGo [2] selects a seed that has higher probabilities to reach the source locations. Several approaches adopt hybrid fuzzing, taint analysis, and machine learning to help fuzzers explore more paths. SAVIOR [8] uses hybrid fuzzing, combining it with concolic execution to explore code blocks guarded by complex branch conditions. RedQueen [1] uses taint analysis and symbolic execution for the same purpose. VUzzer [40] also uses dynamic taint analysis and mutates bytes which are related to target branch conditions to efficiently explore paths. TIFF [18] infers the type of the input bytes through dynamic taint analysis and uses the type information to mutate the input. Matryoshka [7] uses both data flow and control flow information to explore nested branches. In addition to hybrid fuzzing with traditional techniques such as symbolic and concolic executions, NEUZZ [46] adapts neural network and sets the number of covered paths as an objective function to maximize covered paths. Angora [6] adapts both taint analysis and a gradient descent algorithm to improve the number of covered paths. These approaches do not reduce the execution time of each iteration. They are therefore orthogonal to our work. Thus, we can use these approaches to further increase fuzzing performance.

7.2 Optimizing Sanitizers

Since C/C++ programming languages are memory and type unsafe languages, several sanitizers [47] target memory safety violations [5, 23, 41, 48, 49] and type safety violations [14, 19, 24, 29]. Despite their broad use, sanitizers have several limitations such as high overhead, limited detection abilities, and incompatibility with other sanitizers.

To reduce sanitizer overhead, ASAP [52] and PartiSan [25] disable check instrumentation on the hot path according to their policies. The intuition of both approaches is that most of the sanitizer's overhead comes from checks on a few hot code paths that are frequently executed (e.g., instrumentation in a loop). ASAP removes check instrumentation on the hot path based on pre-calculated profiling results at compile time. In PartiSan [25], Lettner et al., propose runtime partitioning to more effectively remove check instrumentation based on runtime information during execution. However, both approaches miss a main source of overhead when reducing the cost of ASan during fuzzing campaigns: the overhead is due to memory management and not due to the low overhead safety checks. As ASAP and PartiSan target the cost of checks, they are complementary to FuZZan. To fuzz quickly, there is an option to generate a corpus from a normal binary, and then feed the corpus to an ASan binary. FuZZan can also adopt this option for fast fuzzing.

Pina et al., [38] use multi-version execution to concurrently run sanitizer-protected processes together with native processes, synchronizing all versions at the system-call level. To synchronize all versions, they use a system-call buffer and a Domain-Specific Language [37] to resolve conflicts between different program versions. Xu et al., [54] propose Bunshin to reduce the overhead of sanitizers and conflicts based on the N-version system through their check distribution, sanitizer distribution, and cost distribution policies. Since these approaches are based on N-version systems, they increase hardware requirements such as several dedicated cores and at least N times of memory. Also, these approaches do not address the fundamental problem of ASan memory overhead.

8 Conclusion

Combining a fuzzer with sanitizers is a popular and effective approach to maximize bug finding efficacy. However, several design choices of current sanitizers hinder fuzzing effectiveness, increasing the runtime cost and reducing the benefit of combining fuzzing and sanitization.

We show that the root cause of this overhead is the heavy metadata structure used by sanitizers, and propose FuZZan to optimize sanitizer metadata structures for fuzzing. We implement and apply these ideas to ASan. We design new metadata structures to replace ASan's rigid shadow memory, reducing the memory management overhead while maintaining the same error detection capabilities. Our dynamic metadata structure adaptively selects the most efficient metadata structure for the current fuzzing campaign without manual configuration.

Our evaluation shows that FuZZan improves performance over ASan 52% when starting with empty seeds (48% with Google's seed corpus). Based on improved throughput, FuZZan discovers 13% more unique paths given the same 24 hours and finds bugs 42% faster. The open-source version of FuZZan is available at https://github.com/HexHive/FuZZan.

Acknowledgments

We thank the anonymous reviewers and our shepherd Julia Lawall for their detailed feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), NSF CNS-1801601, and ONR award N00014-18-1-2674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011.

[5] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[7] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[8] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[9] Google. Address Sanitizer Found Bugs. https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs.

[10] Google. Clusterfuzz. https://google.github.io/clusterfuzz/.

[11] Google. Fuzzer test suite. https://github.com/google/fuzzer-test-suite.

[12] Google. Kernel Address Sanitizer (KASan), a fast memory error detector for the Linux kernel. https://github.com/google/kasan/wiki.

[13] Google. Libfuzzer tutorial. https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md.

[14] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[15] Niranjan Hasabnis, Ashish Misra, and R Sekar. Lightweight bounds checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[16] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Security Symposium (SEC)*, 1992.

[17] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 1988.

[18] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using Input Type Inference To Improve Fuzzing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[19] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[20] Linux kernel document. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[22] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Fast and generic metadata management with mid-fat pointers. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2017.

[23] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[24] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the USENIX Security Symposium (SEC)*, 2015.

[25] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. PartiSan: fast and flexible sanitization via run-time partitioning. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[27] LLVM. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[28] LLVM. The LLVM Compiler Infrastructure Project. http://llvm.org/.

[29] LLVM. TySan: A type sanitizer. https://reviews.llvm.org/D32199.

[30] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *Processings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2001.

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the USENIX Security Symposium (SEC)*, 2019.

[32] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[33] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990.

[34] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[35] NIST. Juliet test suite. https://samate.nist.gov/SARD/testsuite.php.

[36] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[37] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A DSL approach to reconcile equivalent divergent program executions. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[38] Luís Pina, Anastasios Andronidis, and Cristian Cadar. FreeDA: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2018.

[39] The Chromium Project. Address Sanitizer (ASan). https://www.chromium.org/developers/testing/addresssanitizer.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[42] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications (WBIA)*, 2009.

[43] Kostya Serebryany. Hardware Memory Tagging to make C/C++ memory safe(r). https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/HardwareMemoryTaggingtomakeC_C++memorysafe(r)-iSecCon2018.pdf.

[44] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_serebryany.pdf.

[45] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[47] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[48] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[49] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the European Conference on Computer Systems (EUROSYS)*, 2017.

[50] Dmitry Vyukov. Address/Thread/MemorySanitizer Slaughtering C++ bugs. https://www.slideshare.net/sermp/sanitizer-cppcon-russia.

[51] Dmitry Vyukov. Syzbot. https://syzkaller.appspot.com/upstream.

[52] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2015.

[53] Wikipedia. x32 ABI. https://en.wikipedia.org/wiki/X32_ABI.

[54] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing Security Mechanisms through Diversification. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[55] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[56] Yves Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[57] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.

[58] Michal Zalewski. New in AFL: persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations

Chengcheng Xiang
University of California, San Diego

Haochen Huang
University of California, San Diego

Andrew Yoo
University of Illinois Urbana-Champaign

Yuanyuan Zhou
University of California, San Diego

Shankar Pasupathy
NetApp Inc.

Abstract

Configuration has become ever so complex and error-prone in today's server software. To mitigate this problem, software vendors provide user manuals to guide system admins on configuring their systems. Usually, manuals describe not only the meaning of configuration parameters but also *good practice recommendations* on how to configure certain parameters. Unfortunately, manuals usually also have a large number of pages, which are time-consuming for humans to read and understand. Therefore, system admins often do not refer to manuals but rely on their own guesswork or unreliable sources when setting up systems, which can lead to configuration errors and system failures.

To understand the characteristics of configuration recommendations in user manuals, this paper first collected and studied 261 recommendations from the manuals of six large open-source systems. Our study shows that 60% of the studied recommendations describe specific and checkable specifications instead of merely general guidance. Moreover, almost all (97%) of such specifications have not been checked in the systems' source code, and 61% of them are not equivalent to the default settings. This implies that additional checking is needed to ensure the recommendations are correctly applied.

Based on our characteristic study, we build a tool called PracExtractor, which employs Natural Language Processing (NLP) techniques to automatically extract configuration recommendations from software manuals, converts them into specifications, and then uses the generated specifications to detect violations in system admins' configuration settings. We evaluate PracExtractor with twelve widely-deployed software systems, including one large commercial system from a public company. In total, PracExtractor automatically extracts 338 recommendations and generates 173 specifications with reasonable accuracy. With these generated specifications, PracExtractor detects 1423 good practice violations from open-source docker images. To this day, we have reported 325 violations and have got 47 of them confirmed as real configuration issues by admins from different organizations.

1 Introduction

1.1 Motivation

Misconfiguration (error in configuration settings) has become one of the major causes of failures in large-scale cloud and Internet systems, as reported by many system vendors [29,45,64] and service providers [20,26,32,34,37,51]. While various fault tolerance and recovery mechanisms are effective in handling hardware and software failures, they are less effective in handling configuration errors [27,32,37]. In 2017, a configuration error at Level 3, an Internet backbone company, caused a nationwide network outage [22]. On March 13th, 2019, the recent outage in Facebook was also caused by a server configuration error, affecting millions of users [52]. In addition to reliability, configuration errors can also lead to security issues [59]. OWASP reports misconfiguration as one of the top 10 most critical web security risks [38]. In 2017, a configuration error of Amazon S3 storage exposed personal information of 200 million U.S. voters [53].

One of the primary reasons for configuration errors is the ever-increasing configuration complexity, especially with system software [60]. Configuration complexity is partially reflected by the large and almost always increasing number of configuration parameters, as well as their configuration constraints and inter-dependency [31, 35, 44, 47, 63], which inevitably increase system admins' error rates [40, 46]. For example, MySQL 8.0 has more than 460 configuration parameters. Similarly, Apache httpd 2.4 has more than 550 parameters. Such a high level of complexity makes system configuration an error-prone task.

While research efforts have been attempted on reducing configuration complexity [55, 60], it is still a long journey to fully tame the complexity issue. Today, to assist system admins, software vendors typically release user manuals together with their software. A manual describes in detail the name, usages and sometimes constraints of each configuration parameter. It can be in print as a PDF file or accessed electronically as HTML/XML files, providing good guidance

Software	Pages	Software	Pages
COMP-A[1]	8283	Httpd	1009
MySQL	5494	HBase	787
PostgreSQL	3724	Freebsd	726
CentOS	2297	Ubuntu server	413
Hadoop	2331	Zookeeper	181

Table 1: Number of pages in ten popular software's manuals.

and reference for system admins to configure and manage server software.

Unfortunately, system manuals are quite large, containing hundreds or even thousands of pages. Table 1 lists the numbers of pages in the manuals of ten software, including one commercial software, COMP-A [1]. from a large public company. As the table shows, manuals of MySQL, PostgreSQL, CentOS and Hadoop have 2331-5494 pages. COMP-A has 8283 pages in its technical documentation.

With such a daunting number of manual pages, system admins find manuals hard and time-consuming to refer to and understand. As such, system admins often do not refer to them when configuring systems. Instead, they either rely on their own judgment/guesswork or ask for help from other admins [36]. Previous studies have shown that system admins solved only a small proportion of usage problems (4% to 25%) by referring to manuals [21, 33, 36].

However, manuals still contain useful information and ignoring manuals can lead to configuration errors that cause server downtime and data center outages. Figure 1 gives six real-world configuration errors of commercial and open-source software, in which system admins clearly do not follow good practice recommendations in manuals. The misconfigured parameters in these examples were set to incorrect values, leading to problems of systems' availability, performance and security. Since these incorrect values are totally legal values (i.e. violating no constraints in source code), they cannot be detected by software's own checking logic, as well as tools that focus on checking for illegal values [63]. However, in all these cases, the corresponding manuals actually have clearly given recommendations on how to set these parameters. Had these recommendations been followed by system admins, these misconfigurations would have been avoided.

Unfortunately, good practices recommended in manuals or other documents are not fully utilized by system admins to avoid configuration errors mainly due to three reasons:

- Recommended practices are spread out in various parts of manuals and cannot be easily found by system admins due to manuals' bulkiness and poor navigation [36].
- Many good practice recommendations are not always the same as default settings (more details in §2). A recent study shows that admins tend to go with default settings for more than 80% of configuration parameters, and many configuration errors were caused exactly because admins do not change the default setting [60]. As later shown in our evaluation (cf. Table 12), we also found many (997) cases that system admins just went with bad defaults. Had system admins read the recommendations in the manuals, they could have avoided some of these mistakes.
- As shown in all the examples in Figure 1, good practices recommended in manuals are often *soft* constraints, which usually are not checked inside software. Thereby, the violations of them cannot be detected by previous tools that were built by either inferring configuration specification from the software's source code [63] or just directly reusing the source code to check configuration [61].

1.2 Our Contributions

This paper studies the research questions on whether it is useful to automatically extract good practice recommendations from manuals and use them to detect system admins' configuration issues, and if so, how to do it. We first collected and studied 261 recommendations from six large open-source software manuals. Our study shows that 60% of the studied recommendations described specific, checkable specifications instead of just general guidance. In addition, almost all (97%) of the checkable specifications are not checked in source codes, and 61% of them are different from the default settings (reasons and details are discussed in §2).

Based on our characteristic study, we build a tool called PracExtractor, which employs Natural Language Processing (NLP) techniques to automatically extract good practice recommendations from manuals, converts them into specifications, and then uses the generated specifications to detect violations in system admins' configurations.

We evaluated PracExtractor with manuals of *twelve* widely-deployed software systems, including one from a *commercial* company with tens of thousands of customers. Overall, PracExtractor automatically extracts 338 recommendations, with a precision of 86% and a recall of 83%. PracExtractor converts 173 recommendations into specifications with reasonable accuracy. For the six "new" manuals not included in our characteristic study, PracExtractor can achieve a precision of 83% for recommendations and 88% for specifications.

To evaluate the capability of detecting real-world misconfigurations, we run PracExtractor against real-world configurations from top-downloaded container images on DockerHub [24]. PracExtractor detects 1423 violations in 853 images. We reported 325 violations to the image maintainers and got 47 confirmed as real configuration issues, including six issues in images with over 1M downloads and 28 in images with over 1K downloads.

Interestingly, in addition to detecting system admins' configuration problems, PracExtractor also detects a few incorrect

[1] We are required to keep the company and the product anonymous.

Comp-A
Parameter: *export-policy*
Incorrect setting by admins: *none*
Outcome: NFS mount permission deny
Fix: Specify a rule for export-policy
Recommendation in manual:
"Always *create a rule* with a policy to control access to a volume."

(a)

Comp-A
Parameter: *security-style*
Incorrect setting by admins: *mix*
Outcome: NFS volumes are not mounted
Fix: change *security-style* to *unix*
Recommendation in manual:
"Choose either NTFS- or UNIX-style security unless there is a specific reason to use mix."

(b)

Comp-A
Parameter: *rsize*
Incorrect setting by admins: *32k*
Outcome: Performance slowdown
Fix: change *rsize* to *64k*
Recommendation in manual:
"Comp-A highly recommends using *64k* rsize/wsize for better performance with NFS v4."

(c)

Httpd
Parameter: *EnableSendfile*
Incorrect setting by admins: *on*
Outcome: web pages become unavailable
Fix: change the value to *off*
Recommendation in manual:
"Within some filesystems, it is better to *disable* this feature to avoid operational problems."

(d)

HBase
Parameter: *hbase.regionserver.thrift.framed*
Incorrect setting by admins: *false*
Outcome: vulnerable to DoS attack
Fix: change the value to *true*
Recommendation in manual:
"Setting this to false will select the default transport, vulnerable to DoS..."

(e)

Cassandra
Parameter: *enable_transient_replication*
Incorrect setting by admins: *true*
Outcome: potential service failure
Fix: change the value to *false*
Recommendation in manual:
"Transient replication is experimental and is not recommended for production use."

(f)

Figure 1: Six real-world configuration errors that were made by system admins without following recommendations from manuals. (a)(b)(c) are from COMP-A's customer ticket database, (d)(e)(f) are new misconfigurations our work discovered from public Docker images and have been confirmed by multiple image maintainers [3–10].

default settings, three of which have already been confirmed by MySQL and Cassandra developers as real bugs. Incorrect default settings can easily cause configuration errors since system admins are most likely to go with the default [60].

2 Characteristic Study

Before we build a tool to extract good practice recommendations from manuals, we first collected and studied 261 real-world recommendations from manuals of six widely-deployed systems listed in Table 2. Our study answers two questions: (1) *Is it useful to extract those recommendations from manuals?* If they are all general advice such as "recommend to set it to a large value", extracting them is not very helpful since they cannot be used as specifications for automatically checking system admins' settings. In contrast, if the recommendations are clear specifications such as "recommend to set this to greater than 2000", extracting them out from manuals can help build checkers to detect violations to them. Additionally, have developers already put in their code to check if system admins follow these recommended practices? If so, there is no need to extract them from manuals. Finally, how often are these recommendations the default settings for the corresponding configuration parameters? If they are not default, why? *(2) How difficult is it to extract good practice recommendations from manuals?* In particular, are manuals structured enough for information extraction?

Observation 1: *157 (60%) of the studied good practice recommendations are specific instead of just general advice.* We manually studied all recommendations and categorized them based on their contents. If a recommendation is about something that is hard to be checked automatically, it is classified as a "general advice" (e.g. Table 3 last row). Otherwise, it is classified as a "clear specification", which is further categorized into value, usage, correlation, and property by what is recommended, as explained in the caption of Table 2. An example is given for each category in Table 3.

Table 2 shows the number of recommendations of each category. In total, 157 (60%) of the 261 recommendations describe clear specifications that if extracted can be used for automatically checking system admins' configuration settings. The remaining 104 recommendations are general advice that is hard to check automatically.

Observation 2: *152 (97%) of the specific good practices recommended in manuals are not checked in source code.* For each recommendation, we manually examine the source code of each software to see if the recommended practices are checked in source code to warn/inform system admins upon violations. Table 4 shows that only five out of the 157 specific recommendations in manuals are checked in source code.

Listing 1 shows an example where a recommended practice is checked in HBase code. In this case, if the practice is violated by system admins, they will be warned to reexamine the setting of this parameter more carefully.

The goal of our work is exactly to generate more checkings like the HBase case shown in Listing 1, i.e. automatically extract good practice specifications from manuals and build a checker to warn system admins when their settings do not follow the recommended practices.

Violations to good practices may not always be configuration errors. However, as previous work [60, 62] has shown,

Software	#Rec	Specific					General
		value	usage	correl	property	**total**	
MySQL	78	27	6	2	5	40	38
Httpd	92	25	16	8	3	52	40
PostgreSQL	49	21	1	3	3	28	21
HDFS	18	13	0	3	0	16	2
HBase	12	10	1	0	0	11	1
Spark	12	9	0	0	1	10	2
Total	261	105	24	16	12	157	104

Table 2: Characteristics of the 261 studied good practice recommendations from six widely used software. "Specific": describe a clear specification; "value": recommend to (not) set to one or multiple values (e.g. Table 3 row 2); "usage": recommend to (not) use an option, typically for command-line options without a value (e.g. Table 3 row 3). "correlation": recommend to set to a value smaller, larger or equal to another parameter (e.g. Table 3 row 4). "property": recommend to set to a value with some property, such as in the absolute path format(e.g. Table 3 row 5).

Category	Example Practice Description in Manuals
Value	It is generally not desirable to set this to a value *greater than 2000*.
Usage	This option may be useful for diagnostic purposes, to see the exact text of statements as received by the server, but for security reasons is *not recommended for production use*.
Correlation	Setting this *lower than the dfs.namenode.replication.min* is not recommend and/or dangerous for production setups.
Property	It is best to specify the datadir value as *an absolute path*.
General	We recommend that this setting be kept to *a high value* for maximum server performance.

Table 3: Real examples of recommendations of different types.

Software	# (%) of prac checked in code
MySQL	1 (2.5%)
Httpd	1 (1.9%)
PostgreSQL	1 (3.6%)
HDFS	1 (6.3%)
HBase	1 (9.1%)
Spark	0 (0.0%)

Table 4: Number of good practices checked in source code.

```
if(balancedPreferencePercent
< 0.5) {
  LOG.warn("The value of " +
  DFS_DATANODE_BALANCED_SPACE
  _PREFERENCE_FRACTION_KEY +
  " is less than 0.5 so
  volumes with less available
  disk space will receive
  more block allocations");

}
```

Listing 1: Example of a good practice check in HBase's source code.

Software	Same -val	Multi -val	Rela -val	Cond -rec	No default	Others
MySQL	14	10	1	10	4	1
Httpd	16	6	3	9	16	2
PostgreSQL	10	8	2	6	0	2
HDFS	9	1	3	3	0	0
HBase	6	3	0	1	0	1
Spark	6	2	0	1	1	0
Total	61 (39%)	30 (19%)	9 (6%)	30 (19%)	21(13%)	6 (4%)

Table 5: The number of recommendations that are the same as the default and different categories of recommendations that are not the same as the default (multiple-value, relative-value, condition-recommendation, no-default, and others).

many system admins simply rely on guesswork or unreliable sources (e.g. online forums) to configure complex server software. If our checker can give a warning like Listing 1 when the settings do not follow practices, system admins can at least have a chance to *reexamine* the settings more carefully.

Observation 3: *96 (61%) of the specific good practice recommendations are not equivalent to default settings.* It is conceivable that some recommended practices might be the default settings (after all, the vendor recommends them). If this is the case, there is no need to extract recommended practices from manuals. System admins simply just go with default if they do not know how to set it better.

However, as shown in Table 5, only 61 (39%) good practices are equivalent to the default settings. For the majority (61%) cases, recommendations are not the same as default due to several reasons, including (a) 30 recommend multiple different values, e.g. a range or a set of values. In real settings, they may need to be modified to accommodate different situations, so it is worthwhile for sysadmins to double-check if the settings follow recommendations.; (b) 30 recommend some settings based on some conditions, e.g. "Enable A along with B"; (c) 21 recommendations are on command line options that have no default values; (d) 9 recommend relative values, such as "25% system RAM size"; (e) 6 cases have no clear reason why the default is different. They may be potential bugs and we have one of them confirmed as a bug by developers.

Observation 4: *The six studied manuals are organized in a similar structure.* As shown in Table 6, the six manuals are either in HTML or XML format and parameters in them are described in a similar structure:

- Each parameter is described in one separate section.
- Parameter names are often used as the section headings.
- There is some meta-info of the parameter described in the format of `<key>:<value>`, such as "Type:string".
- Most information related to each parameter is described in one or several paragraphs of free texts.

The per-parameter section structure makes it possible to relate each parameter name and its description by parsing the section structure. In addition, data types and default values can be used to identify parameter values in plain text descriptions which is necessary for generating specifications.

3 Design and Implementation

We design and implement PracExtractor to automatically extract recommendations from manuals, convert them into specifications and then uses them to detect violations. PracExtrac-

Software	Manual format	Parameter section?	Data type?	Default value?
MySQL	html	Yes	Table	Table
Httpd	xml	Yes	No	Table
PostgreSQL	html	Yes	KV	Text
HDFS	xml	Yes	No	Table
HBase	html	Yes	No	KV
Spark	html	Yes	No	Table

Table 6: Format and structure of manuals regarding how they describe configuration parameters. "Parameter section"— a separate section describes each individual parameter, "Data type"/"Default value" — the format they are described in, including table and KV (`<key>:<value>`).

tor faces two main challenges: (1) As manuals are written in plain texts and have a large amount of texts unrelated to recommendations, how to effectively filter noises and extract only recommendations? (2) Even after we extract recommendations, how to convert them into formal specifications that can be used to automatically check for violations?

To address the first challenge, PracExtractor breaks manual texts into sentences and extracts recommendation sentences with two filtering steps: keyword-based filtering (coarse grained) and syntactic-pattern-based filtering (fine grained). PracExtractor mines the keywords and syntactic patterns from the studied 261 recommendations.

To address the second challenge, PracExtractor first identifies semantic entities (e.g. parameter name and values) in a recommendation sentence and then convert it into a formal specification by matching them with semantic patterns.

3.1 Preprocessing and Parsing

PracExtractor first preprocesses and parses software manuals into parameter name, meta-info and free-text descriptions. The meta-info, including type and default value, is necessary for recognizing setting entities later (cf. §3.3). One special parameter type is enum, for which manuals usually also indicate all valid values along with a parameter. PracExtractor extracts the valid values for each parameter and uses them to identify enum values from a sentence in §3.3.

PracExtractor parses manuals based on the observed manual structure. Table 6 shows that manuals are usually written in HTML/XML formats with separate sections for different parameters. PracExtractor parses HTML and XML files, identifies each separate parameter section, and extracts parameter name, meta-info and free-text description from each. PracExtractor then breaks free-text descriptions into sentences.

Different manuals may still have slightly different formats for parameter sections. To handle that, PracExtractor takes an input of a small code snippet (format spec). A format spec is easy to write: according to our evaluation of twelve large manuals, they are typically fewer than 30 lines of Python code, and each of them can be written in 0.5-2 hours. (cf. Table 14).

Word	Covered sentences
recommend	74
well	26
good	26
appropriate	21
want	17

Bigram	Covered sentences
be recommended	34
should only	20
may want	13
good idea	7
with caution	7

Table 7: 10 sample keywords (words and bigrams) collected by PracExtractor from 261 studied recommendations and how many recommendations each covers.

3.2 Recommendation Sentences Extraction

Most sentences in manuals do not contain recommendations. For the twelve evaluated software manuals, 696–25510 sentences are extracted from parameter sections, but only 0.4%–2.7% of them contain recommendations. To extract these small percentage of recommendations, PracExtractor performs two steps filtering:

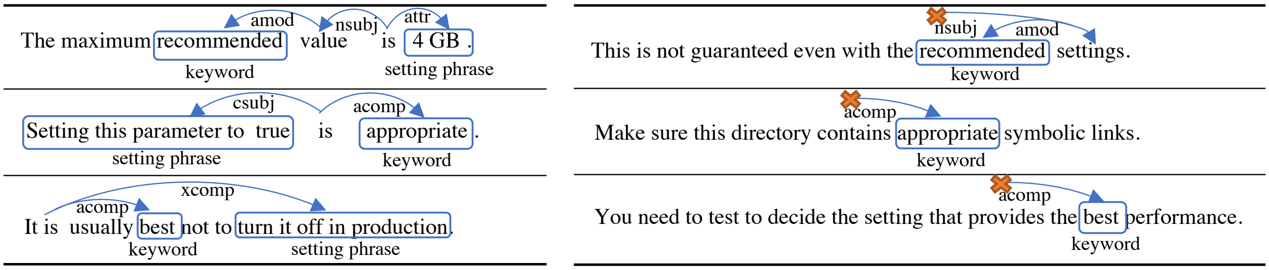
Keyword-based Filtering Following the intuition that recommendations are usually described with certain keywords (e.g. "recommend", "suggest"), PracExtractor extracts candidate recommendations with keyword filtering. To find out which words/phrases should be used as the keywords, PracExtractor first breaks the studied 261 recommendation sentences into individual words and bigrams (two consecutive words) and uses them as the candidate keywords T. PracExtractor then uses inverse document frequency (IDF) to rank the candidate keywords. IDF reflects how frequently a term t (word/bigram in our case) occurs in a set of sentences set S, as:

$$IDF(t,S) = \log\frac{|S|}{|\{s \in S : t \in s\}|}.$$

PracExtractor calculates $IDF(t,R)$ for the studied recommendations R and $IDF(t,S)$ for all the manual sentences S. PracExtractor ranks T based on $IDF(t,R)$ and $IDF(t,S)$ and get the smallest 100 and 300 terms separately as T_R and T_S. PracExtractor uses $T_R - T_S$ as the final keywords. The intuition behinds this is to find the words that are important in recommendations but not normal sentences. In Table 7, the sample keywords show that PracExtractor has effectively found keywords related to recommendations.

Syntactic-Pattern-based Filtering Using keyword filtering alone is not enough. After keyword-based filtering, only 7.3% of the remaining sentences are recommendations. Many sentences with the recommendation-related keywords are not true recommendations. Figure 2 (a) and (b) gives examples that the same keywords can be contained both in recommendation and non-recommendation sentences.

The key difference between these recommendations and non-recommendations in Figure 2 is their syntactic patterns.



The maximum recommended value is 4 GB .

Setting this parameter to true is appropriate.

It is usually best not to turn it off in production.

This is not guaranteed even with the recommended settings.

Make sure this directory contains appropriate symbolic links.

You need to test to decide the setting that provides the best performance.

(a) Recommendation sentences

(b) Non-recommendation sentences

Figure 2: Comparison of syntactic patterns of recommendation and non-recommendation sentences that contain likely-recommendation keywords. The patterns are labeled as undirected dependency paths from a keyword to a setting phrase, where a dependency path consists of a sequence of syntactic relations annotated with Universal Dependencies [23]: **amod** – link from a noun to an adjective modifier; **nsubj** – relation between a verb/noun and a prepositional phrase; **attr** – relation between a verb/adjective and a complement, etc.

Besides a keyword, the recommendations also contain a *setting phrase*, a noun/verb phrase describing what setting is recommended. Between such setting phrases and keywords, there are certain syntactic relations (patterns), which do not exist in non-recommendation sentences. PracExtractor leverage the syntactic-patterns to do fine-grained filtering.

PracExtractor first adopts the universal dependency (UD) tree [23] to represent a sentence's syntactic structure. A UD tree $T = (V, E)$ consists of vertices V and edges E, where $v \in V$ is labeled with a word's part of speech (POS) and $e \in E$ represents the syntactic dependency between two words (cf. Figure 2). Let $T' = (V, E')$ be an undirected correspondence of T, the syntactic pattern between a keyword and a setting phrase can be represented with an undirected path $p = (v_0, e'_{v_0,v_1}, v_1, \ldots, v_n)$, where v_0 is the keyword, v_n is the setting phrase, and $e'_{v_{i-1},v_i} \in E'$ for $i \in [1, n]$.

With the UD representation, PracExtractor mines the unique patterns for recommendations from the studied 261 recommendations S_{rec} and a set of non-recommendation samples $S_{\text{not\_rec}}$ that contains the keywords. For each sentence s, PracExtractor builds $T'_s = (V_s, E'_s)$ and extracts all paths

$$\begin{aligned}\rho_s = \{(v_0,\ e'_{v_0,v_1},\ v_1,\ \ldots,\ v_n) : \\ v_0 \in \textsc{Keywords} \wedge \forall i \in [1,n]\ e'_{v_{i-1},v_i} \in E'_s \\ \wedge \forall i \in [0,n]\ v_i \in V_s \wedge v_n \in \textsc{SettingPhrases}\},\end{aligned}$$

that starts from each keyword and ending at each setting phrase. The keywords are from the last step and the setting phrases are labeled by human inspectors. PracExtractor extracts all such paths from all recommendations and non-recommendation samples, denoted as P_{rec} and $P_{\text{not\_rec}}$. PracExtractor then extracts patterns P_{pattern} with Algorithm 1.

With the identified syntactic patterns P_{pattern}, PracExtractor classifies a new sentence s' into a recommendation or non-recommendation. PracExtractor traverse P_{pattern} and check if any pattern matched with s'. If at least one pattern matched then s' is classified as a recommendation otherwise non-recommendation. Such a matched pattern also labels the setting phrase in s' at the pattern's end (cf. Figure 2). The setting phrase will be used in specification generation (cf.§3.3).

Algorithm 1: Syntactic-pattern extraction algorithm

```
Input: $P_{\text{rec}} = \bigcup_{s \in S_{\text{rec}}} \rho_s$, $P_{\text{not\_rec}} = \bigcup_{s \in S_{\text{not\_rec}}} \rho_s$
Output: a pattern set $P_{\text{pattern}}$
$P'_{\text{rec}} \leftarrow []$, $P_{\text{pattern}} \leftarrow \emptyset$;
for $\rho_i \in P_{\text{rec}}$ do
    // Collect all the prefixes of $\rho_i$
    for $\rho_{i,j} \in \texttt{prefix}(\rho_i)$ do
        $P'_{\text{rec}}.\texttt{append}(\rho_{i,j})$;
// Traverse elements in $P'_{\text{rec}}$ in the order of frequency
// to extract the most general patterns
for $\rho_i \in \texttt{mostFrequentElement}(P'_{\text{rec}})$ do
    if $\texttt{prefix}(\rho_i) \cap P_{\text{pattern}} \neq \emptyset$ then
        continue;
    if $\rho_i \notin P_{\text{not\_rec}}$ then
        $P_{\text{pattern}} = P_{\text{pattern}} \cup \{\rho_i\}$;
return $P_{\text{pattern}}$
```

3.3 Specification Generation

In §3.2, PracExtractor identifies recommendation sentences and the setting phrases within it. PracExtractor then converts the setting phrases into checkable, formal specifications. Table 8 gives three example recommendations and the corresponding specifications. In general, PracExtractor can generate four types of specifications, including value, correlation, usage and property, as shown in Table 9.

A naïve way to generate specifications is to match setting phrases with predefined regular expressions and convert them accordingly. This can transform simple phrases with numbers (e.g. "less than 8"), but cannot convert more complex phrases (e.g. phrases with `enum` or parameter names). For instance, a phrase could be "set to chain", where "chain" is an `enum` value in Httpd. Such software-specific words can hardly be predefined in regular expressions and so cannot be matched.

Sentence	Specification
It is recommended to enable this option.	p == true
A value between 8 to 16 is suggested.	p ∈ [8, 16]
We suggest to set it less than ThreadsPerChild.	p < ThreadsPerChild

Table 8: Examples of specifications generated by PracExtractor. Setting phrases are marked with rectangles.

Category	Specification	Description Patterns Example
value/ correlation	p == v p < v | p > v p ∈ [v, v′] p ∈ {v, v′}	v$_{\text{<value>}}$ less$_{\text{syn}}$ | more$_{\text{syn}}$ than v$_{\text{<value>}}$ between$_{\text{syn}}$ v$_{\text{<value>}}$ to v′$_{\text{<value>}}$ v$_{\text{<value>}}$ or v′$_{\text{<value>}}$
correlation	with (p, p′) prefer (p, p′)	along$_{\text{syn}}$ with p′$_{\text{<para>}}$ prefer$_{\text{syn}}$ p′$_{\text{<para>}}$
usage	use (p)	used$_{\text{syn}}$ | useful$_{\text{syn}}$
property	format (p, f)	f$_{\text{<format>}}$

Table 9: Category of specifications PracExtractor generates and example of patterns for each specification. "<value>" is defined as "<bool>\|<num><unit>?\|<enum>\|<parameter>" from Table 10. "less$_{\text{syn}}$" means the synonyms of "less".

PracExtractor addresses this issue in three steps. First, given a recommendation sentence, PracExtractor identifies which parameter the sentence is associated with. Then, PracExtractor uses the parameter's meta-info (e.g. type and default value) extracted before (cf. §3.1) to identify setting entities, such as values and formats. Third, PracExtractor matches the identified setting entities with predefined semantic patterns to generate specifications.

Identify Parameter Names PracExtractor first identifies which parameter a sentence is associated with. For a recommendation sentence s in a paragraph p related to parameter X, there are four possible cases: 1) Only X is mentioned in s. PracExtractor determines this sentence is for X; 2) Another parameter Y is mentioned in s. PracExtractor checks if Y is a subject or object to a verb like "set" or "specify" and determines the sentence is for Y if it is the case; 3) No parameter is mentioned in s. PracExtractor further searches previous sentences in paragraph p; 4) No parameter is mentioned in p, PracExtractor determines the sentence is for X.

Identify Settings Entities Give the identified setting phrase and associated parameter of a sentence, PracExtractor then recognizes setting entities (e.g. values and formats) from the setting phrase based on the associated parameter's type. Table 10 shows the seven types of setting entities that PracExtractor can identify. For different types of setting entities, PracExtractor identifies them with different syntax:

Type	Setting Syntax
<bool>	"enable" | "on" | "true" | "disable" | "false" | "off"
<num>	[-+]?\d+(\.\d+)?
<unit>	"byte" | "MB" | "ms" | "%" | "% of RAM" | …
<enum>	∀w ∈ VALID_VALUES
<parameter>	∀w ∈ ALL_PARAMETERS
<format>	"email address" | "absolute path" | "domain name" | …
<string>	∀w ∉ (<bool> ∪ <num> ∪ <unit> ∪ <enum> ∪ <parameter> ∪ <format>)

Table 10: Types of setting entities PracExtractor identifies from recommendation sentences. `VALID_VALUES` and `ALL_PARAMETERS` are from the type-info and parameter list that PracExtractor identifies in the parsing step (cf. §3.1).

- For basic types, including `<bool>` and `<num>`, PracExtractor identifies them with regular expressions. For `<num>`, PracExtractor also identifies common `<unit>` (e.g. "GB", "byte") along with it.
- For `<enum>`, PracExtractor takes advantage of parameters' type-info to identify it. The type-info of an `<enum>` parameter does not only indicate it is `<enum>` but also indicates the valid values that can be set to the parameter. PracExtractor has parsed these in the parsing step (cf. 3.1) and now searches a setting phrase for the valid values.
- `<parameter>` is for the case that the setting phrase describes current parameter with respect to another parameter, such as "set A to be larger than B". PracExtractor identifies this by searching for valid parameter names in the setting phrase. Note PracExtractor has extracted all parameter names in the parsing step (cf. 3.1).
- For `<format>`, PracExtractor identifies common formats (e.g. "email address", "absolute path") of parameters based on word matching. PracExtractor allows users to provide new words to extend the identifiable formats.
- All other words/phrases are identified as `<string>`. PracExtractor further handles two kinds of strings that have special meaning. First, some of them use "this value" or "this" to refer to a value mentioned in previous sentences. PracExtractor recognizes such references and identifies the actual value from previous sentences. Second, some may use "default" to refer to parameters' default value. In this case, PracExtractor uses the corresponding default value extracted from the parsing step (cf. §3.1) as the recommended setting.

Generate Specification PracExtractor generates checkable formal specifications by matching the setting phrases with predefined semantic patterns. Table 9 lists the types of specifications that can be generated by PracExtractor, each with an example pattern. Before blindly matching a setting phrase with patterns, PracExtractor first considers the associated parameter's data type. For example, if the data type is `<bool>` or `<enum>`, there is no need to match semantic patterns like "less than `<value>`", "between `<value>` and `<value>`", or anything that is for `<num>`. Second, synonyms are also considered for these patterns. For instance "less$_{syn}$ than" can also match with "lower than" and "smaller than", etc.

Detect Negation PracExtractor also detects negation in a recommendation sentence and negates a specification when necessary. Two different types of negations are handled by PracExtractor. First, PracExtractor finds direct negation words, such as "not" (or abbreviation "n't") "none" and "never", in sentences. In addition, PracExtractor also detects indirect negation words and phrases, such as "avoid", "with caution", "rarely", and "seldom", etc. to identify the negation.

3.4 Violation Detection from Configurations

PracExtractor parses a configuration file and turns the settings into key-value pairs of (`parameter name`, `value`). Although the formats of configuration files for different software can vary depending on the software implementation, most of them have similar formats. The configuration files of the twelve popular systems in our evaluation, including MySQL, Httpd, HBase, HDFS, Spark, Squid and even the commercial system, all follow two common and simple formats: key-value pairs with separator like '=' or ':' or XML format.

Then PracExtractor checks the parsed parameter values against extracted specifications and generates warning messages if it detects violations. PracExtractor uses the *original sentences* from manuals as the warning messages. An example warning could be "Setting dfs.safemode.replication.min lower than dfs.replication.min is not recommended and is dangerous for production setups". This warning can remind sysadmins to double-check the configurations to avoid potential mistakes, just like the one shown in Figure 1 and 3.

Most of the violations detected by PracExtractor cannot be detected by previous works. Previous works [44, 61, 63] mainly use configuration checking/usage logic in source code to detect misconfigurations. However, most recommendations (97% as in Table 4) are not checked in source code, so violations to them cannot be detected by these works.

4 Experimental Evaluation

As shown in Table 11, we evaluate PracExtractor on twelve large software systems including eleven popular open-source server systems and one commercial systems (COMP-A) from a public company that serves many enterprise customers. These systems' manuals have 543 to 8283 pages. These manuals include not only the six manuals in our characteristic study, but also six new manuals that are not studied before. We evaluate both the precision and recall of PracExtractor by comparing its results with recommendations identified by human inspectors. In our evaluation, two human inspectors manually and independently examined each manual to identify recommendations.

For violation detection, we evaluate PracExtractor with real-world settings from top-ranked container images on DockerHub (200 images for each open-source systems). We run PracExtractor against configuration files in these docker images to detect violations to the specifications extracted from manuals. The evaluated images include both Linux-based and Windows-based one. However, as PracExtractor currently does not support platform-specific checking, our evaluation does not include checking platform-related specifications. In total, only 4 specifications are platform-related.

Recommendation and Specification Extracted As shown in Table 11, PracExtractor extracts a total of 338 good practice recommendations (including specific and general advice) from manuals and automatically converts 173 of them into formal specifications that can be used to check system admins' configuration settings. Among all software, Httpd has the most number (81) of recommendations extracted as well as the most number (31) of specifications generated.

Results with the six "new" manuals excluded from our study PracExtractor works reasonably well with the six "new" software manuals that are not included in our characteristic study. PracExtractor extracts 117 recommendations and 59 specifications from these manuals. For example, it extracts 35 recommendations and 22 specifications for the commercial software, COMP-A. The precision and recall are only slightly lower than the one for the six studied manuals, but are still reasonably good (with a 0.83 precision and 0.80 recall for recommendations, and 0.88 precision and 0.66 recall for specifications).

Violations Detected PracExtractor detects in total 1423 practice violations from 853 unique images. We manually validated all the violations. We reported 325 (that are maintained on GitHub) of them to their maintainers and have got 47 confirmed as real configuration issues, including six in images with >1M downloads and 28 in images with >1K downloads.

Table 12 shows a breakdown of all detected violations. 426 are "wrong change", namely a parameter is explicitly changed to a non-recommended value by system admins. 997 violations are "wrong default", namely a parameter has a non-recommended default value but is not changed. This also matches with the finding from a previous real-world misconfiguration study [60] that many configurations are left as default. In this case, system admins are afraid of changing default settings, even though the default is not recommended or is simply a placeholder value, which needs system admins to explicitly change to fit their own system environments.

Software	Category	Update Time of Manuals	# Recommendations				# Specifications			
			total	extracted	precision	recall	total	generated	precision	recall
MySQL	database	Aug. 2019	78	61	0.90	0.78	40	30	0.88	0.75
Httpd	web server	Aug. 2019	92	81	0.83	0.88	52	31	0.79	0.60
PostgreSQL	database	Aug. 2019	49	38	0.95	0.78	28	20	0.87	0.71
HDFS	distributed storage	Aug. 2019	18	17	1.00	0.94	16	14	0.93	0.88
HBase	distributed storage	Aug. 2019	12	12	1.00	1.00	11	11	1.00	1.00
Spark	distributed computing	Aug. 2019	12	12	0.86	1.00	10	8	0.89	0.80
COMP-A	commercial storage	May 2019	49	35	0.70	0.71	37	22	1.00	0.59
Nginx	proxy	Jul. 2019	26	24	0.92	0.92	6	4	0.50	0.67
Flink	stream processing	Aug. 2019	10	6	0.67	0.60	6	4	1.00	0.67
Squid	proxy	Feb. 2019	22	18	0.86	0.82	13	9	0.82	0.69
Mapred	distributed computing	Aug. 2019	25	20	0.95	0.80	15	9	1.00	0.60
Cassandra	distributed storage	Aug. 2019	15	14	0.93	0.93	13	11	0.85	0.85
studied			261	221	0.89	0.85	157	114	0.87	0.73
new			147	117	0.83	0.80	90	59	0.88	0.66
overall			408	338	0.86	0.83	247	173	0.87	0.70

Table 11: Numbers of recommendations extracted and specifications generated by PracExtractor and corresponding accuracy (precision=$\frac{\text{TruePositive}}{\text{TruePositive+FalsePositive}}$ and recall=$\frac{\text{TruePositive}}{\text{TruePositive+FalseNegative}}$). Recommendations consist of both general advice and specific ones that can be converted into specifications. "studied" refers to the software included in our characteristic study (first 6 rows) , while "new" refers to other software not included in our study (last 6 rows) .

Software	Wrong change	Wrong default	Software	Wrong change	Wrong default
MySQL	20	200	Nginx	0	0
Httpd	338	200	Flink	0	0
PostgreSQL	8	0	Squid	20	0
HDFS	21	0	Mapred	0	0
HBase	0	199	Cassandra	9	398
Spark	10	0			
Total	Wrong change	426	Wrong default	997	

Table 12: Detected good practice violations in container images from Dockerhub. We reported 325 violations to the image owners, and 47 of them have been confirmed as real configuration errors. Three wrong defaults are also confirmed by MySQL and Cassandra.

Figure 3 gives three examples of real-world violations that are detected by PracExtractor from popular container images on DockerHub. They have been confirmed by the image owners as real configuration errors or by the software developers as real bugs. Here are the root causes:

(a) HDFS manual recommends to leave the parameter as `true` to avoid registration of excluded hostnames. However, it is ignored and violated in 21 images, which can cause security issues. We reported them and so far two of them have been confirmed [11, 12].

(b) Cassandra manual does not recommend to enable the experimental feature as it may cause potential failure. However, the default setting enables it, which is a bug, and 199 images just keep the default. The bug has been confirmed and fixed by Cassandra in its new version [1].

(c) The default setting for this parameter in MySQL is much larger than the recommended value in the manual, and the default value (set by MySQL developers) is actually incorrect. We report it to MySQL official Bugzilla and it has been confirmed as a bug [16].

Indeed, not all the 1423 violations are configuration errors or bugs. However, as discussed before in the real-world example from HBase (cf. §2 Listing 1) that explicitly performs such checks in its source code, when such violations are warned to system admins, they at least get a chance to *reexamine and reconsider* the settings more carefully.

Maintainers' Feedback on Violations We reported 325 to the image maintainers and so far have got 47 violations confirmed that they need to be changed. We list three example confirmations in Table 13 (row 1, 2, 3). We took a further look into the impact of these confirmed violations: 11 cause security vulnerabilities, 31 cause unreliable services, 2 cause performance issues and 3 cause database inconsistency.

We also got 46 other feedbacks that the maintainers hesitated to fix the violations. They either think it is the upstream vendors' responsibility to handle the issues (Table 13 row 4, 5) or are aware of the limitations but make the settings for particular environments (Table 13 row 6).

Lines of Customized Code for Each Manual Table 14 shows the lines of Python code (LOC) of the format spec for each manual. It needs only 6-73 LOC for the software manuals with hundreds and thousands of pages. Also, the LOC is not proportional to the number of pages in manuals, and it is only one-time effort to each software. On average, it needs little effort (0.5-2 hours) to customize PracExtrac-

HDFS

Parameter: *dfs.namenode.datanode.registration.ip-hostname-check*
Default: *true*
Recommendation: *true*
Misconfiguration: *false*
Violated docker images:
Babbleshack/hadoop, jamesmcclain/hadoop

Recommendation in manual:
"It is recommended that *this setting be left on* to prevent accidental registration of datanodes listed in the excludes file…"

(a)

Cassandra

Parameter: *enable_materialized_views*
Default: *true*
Recommendation: *false*
Misconfiguration: keep wrong default
Violated docker images:
199 images

Recommendation in manual:
"Materialized views are considered experimental and are not recommended for production use."

(b)

MySQL

Parameter: *max_binlog_cache_size*
Default: *2^64*
Recommendation: *4GB*
Misconfiguration: keep wrong default
Violated docker images:
200 images

Recommendation in manual:
"The maximum *recommended value is 4GB* …MySQL currently cannot work with binary log positions greater than 4GB."

(c)

Figure 3: Example of new violations detected by PracExtractor in popular images from DockerHub. The violations have been confirmed by image maintainers [11, 12] and software vendors (Cassandra and MySQL). Cassandra has fixed (b) in its new version [1].

Image	Feedback on Violation Reports from Image Maintainers
eviles/ httpd	"Ok, I've fixed two images: eviles/httpd, eviles/httpd-tomcat.' [3]
newnius/ hbase	"Thanks for pointing out this. I have added that to the default configuration files and rebuilt the images." [5]
oscerd/ cassandra	"Yes we can do that. I can update the configuration for 3.10 and 3.11." [9]
vitessio/ mysql	"As a general strategy, I plan to use MySQL's default values unless there is a strong use-case to override." [13]
madflojo/ cassandra	"Since we are using the upstream cassandra/latest, I'd prefer that this issue go to them." [14]
publicisw/ httpd	"We are aware of these limitations...This is only used on new linux kernels, which should support this feature..." [15]

Table 13: Example of positive and less-positive (in gray background) feedback. Due to page limit, we list three for each.

tor for a new software manual, including even for the large commercial software manual with 8283 pages.

Accuracy — False Negatives and False Positives Table 11 shows PracExtractor's accuracy in terms of recall and precision. For recommendation extraction, PracExtractor has a reasonable high recall, 0.83. In other words, overall, PracExtractor misses only 17% of the recommendations. For specification extraction, PracExtractor's recall is slightly lower (0.70) (i.e. miss 30% of the specification). This is mainly because some descriptive texts and string values are hard to be automatically recognized and converted into specification (cf. Table 15). This can be further improved by analyzing the semantics of parameter names so that string values can be better matched with parameter meanings.

PracExtractor has low false positives, too. Its overall precision is 0.86 for recommendation and 0.87 for specification. That is, only 14% of the recommendations and 13% of the specifications extracted by PracExtractor are false positives. The false positives are introduced mainly due to texts are incorrectly identified as parameter values (cf. Table 15).

Software	Manual pages	LOC
MySQL	5494	73
Httpd	1009	10
PostgreSQL	3724	24
HDFS	1031	12
HBase	787	14
Spark	599	6

Software	Manual pages	LOC
COMP-A	8283	18
Nginx	543	24
Flink	6152	6
Squid	1391	10
Mapred	1318	12
Cassandra	913	11

Table 14: Lines of code (LOC) of format specs for the twelve evaluated manuals.

Impacts of False Positives The false positives will not cause serious impacts as they can be recognized easily. Table 16 shows three cases of false recommendations that PracExtractor extracted from the evaluated manuals. They *are* descriptions related to the parameters but just are not recommendations. For example, "there may be circumstances where it is desirable for a configuration section's authorization to be combined with that of its predecessor" is a false positive, which describes a parameter usage but gives no recommendation. Since PracExtractor includes such an *original* sentence from manuals in a warning message, sysadmins who read it can easily realize that it is not a recommendation and will not be misguided.

Evaluation with Existing Misconfiguration Dataset We also evaluate PracExtractor with existing configuration issues. As there is no existing dataset for good practice violations, we use a dataset [2] for general configuration issues used in previous works [60, 61]. This dataset contains configuration issues from various online forums and mailing lists. We use the issues categorized as "error" to evaluate PracExtractor. Out of the 63 evaluated configuration errors, PracExtractor can detect 7 (10%) of them. We validated that these detected errors cannot be detected by previous works [35, 61] as they violate no constraint in source code while previous works use code constraints to detect errors. For the undetected errors, we found that manuals do not provide recommendations for

Software	False Negatives		
	R1	R2	R3
Httpd	10	11	0
COMP-A	8	3	4
Mapred	1	2	3

Software	False Positives	
	R4	R5
Httpd	4	4
Nginx	4	0
Squid	2	0

Table 15: Root causes for PracExtractor's False Negative and False Positive. R1 — descriptive recommendations that are not identified, such as "it is recommended to not configure a ticket key file". It is hard to automatically infer that this refers to not using `SSLSessionTicketKeyFile`. R2 — unknown string values that are not identified. For example, in "it's recommended the username 'anonymous' is in allowed userIDs", it is hard to recognize the common word "anonymous" as a value. R3 — sentences that are not covered by our syntactic patterns. R4 — texts that are incorrectly identified as parameter values. R5 — non-recommendation sentences that are mismatched with syntactic patterns of recommendations.

the error-related parameters. With a grain of salt, this shows that current manuals have not provided enough recommendations for avoiding many real-world misconfigurations. Further enriching manuals with good practices may improve PracExtractor's detecting capability and also benefit system admins who refer to manuals to resolve configuration issues.

5 Discussion

Generality. PracExtractor is reasonably general for manuals from different software. As our evaluation on six new manuals shows, PracExtractor can identify most (80%) recommendations from these manuals with a precision of 83%.

Human Effort. PracExtractor can easily be extended with new format specs to accommodate future manuals. In our evaluation, the specs for the six new software manuals are all less than 30 lines (cf. Table 14) and were written by a first-year graduate student, each in 0.5-2 hours.

Accuracy of Manuals. Another concern is whether manuals themselves deliver accurate information for PracExtractor to extract. After all, manuals can be outdated and can have mistakes made by the writers. In our work, we considered these factors. First, we validated the last update time of our evaluated manuals. As shown in Table 11, all twelve manuals are updated recently this year. Second, we compared the specifications extracted from manuals against source code and reported all differences to developers to check. If either is wrong/obsolete, it may introduce problems. Interestingly, in three cases, developers from MySQL and Cassandra confirmed that the source code is wrong (cf. Figure 3).

6 Related Work

Misconfiguration detection and troubleshooting. Many works have been done on detecting [25, 28, 30, 61, 66, 68] and troubleshooting [17–19, 42, 43, 54, 56, 57, 65, 69] configuration errors. Almost all of these works detect configuration errors by either checking against (a) patterns mined from tons of configuration files, or (b) constraints inferred from source code. Our work is complementary to these approaches. We extract recommendations from vendor-provided manuals as specifications, use them to detect violations, and warn system admins to reexamine the violations. Each of the approaches has its own strengths and weaknesses. Below, we compare our approach with each previous one respectively.

Parameter	False Positive Recommendations
adaptive_hash _index	It may be desirable to dynamically enable or disable adaptive hash indexing to improve query performance.
AuthMerging	There may be circumstances where it is desirable for a configuration section's authorization to be combined with that of its predecessor.
LimitRequest Line	When name-based virtual hosting is used, the value for this directive is taken from the default (first-listed) virtual host best matching the current IP address and port combination.

Table 16: Example of false positive recommendations PracExtractor extracted. They can be easily recognized by system admins and will not misguide them to make wrong changes.

Xu [61,63], Rabkin [44], and Nadi [35] propose approaches to extract configuration constraints from source code using static analysis. While it can infer simple constraints such as parameter types, range and some simple dependencies, it is less effective in checking against more complex constraints, especially *configuration settings that are legitimate but may not be good or optimal*. In our work, we focus on good practices recommended by vendors. If a system admin configures a parameter with a valid setting but does not achieve the intended goal (e.g performance, reliability or security goal), previous works that focus on detecting *invalid* configurations will not report any problem. In comparison, our PracExtractor can still warn him/her about the setting if it does not follow the good practices PracExtractor extracts from manuals.

Encore [68], PeerPressure [54] and Santolucito et al's work [47, 48] propose to extract configuration constraints and good practices from existing settings. These approaches assume that a large number of *independent* configuration samples are available for learning and the correct configurations are the common ones. This assumption can be true for some systems where configuration settings can be collected from users/customers back to vendors. However, for many enterprise software such as database or storage systems that are mainly deployed in enterprises (e.g. financial companies and government), each system's configuration settings are confidential information and cannot be shared back with vendors. As such, these approaches are less applicable. In comparison, PracExtractor is applicable to such scenarios because it automatically extracts good practice recommendations that *are already written in vendor-provided manuals*, and the checkers generated by PracExtractor can be shipped to enterprise customers to check their configuration settings.

In addition to the above two approaches, another closely related work is ConfSeer [41], which takes a user's configuration of one or multiple parameters to search against the vendor's Knowledge Base (KB) articles and identifies those highly relevant ones so that users can read those KB articles to self-diagnose and self-correct misconfiguration (with no need to call customer support). While this work also uses NLP techniques, their goal is to narrow down the match so that users do not need to read hundreds of KB articles. For a given configuration parameter setting from a user, ConfSeer returns *a ranked list of KB articles for the user to explore, in a way similar to a search engine like Google that tries to return the most relevant web pages to a user's query*. In other words, ConfSeer is an improved Google search engine for configuration-related KB articles. In comparison, our goal is to extract configuration recommendations from manuals and convert them into *formal and checkable* specifications. Our checker can be shipped to the customer sites to *automatically* detect violations and warn system admins to reexamine their settings to proactively avoid problems instead of waiting for postmortem troubleshooting.

Inferring specification from text. Some past works also aim to infer specifications from program-related texts, including from program comments [49, 50], API documents [39, 67, 70, 71], and man pages [58]. Our work differs from previous works both on purposes and techniques. First, instead of helping developers find bugs in source code as in [49, 50, 70], our work aims to help system admins detect misconfigurations in their system settings. Secondly, comments and API documents have relatively uniform structures, which makes it easy to extract information like function names and variable types. In comparison, manuals are much less structured. The only structure is that each parameter has its own section/chapter. Inside a section, it is mostly free text. Thirdly, PracExtractor extracts much more complex constraints than previous work on man pages [58]. DASE [58] uses regular expression to extract valid options from man pages. In comparison, PracExtractor can extract option value, correlation and property from software manuals.

7 Conclusions and Future Work

This paper focused on the usefulness and feasibility of extracting good practice recommendations from software manuals to detect configuration problems. Specifically, we first conducted a characteristic study on 261 recommendations from six large open source software manuals. Based on the observations learned, we designed and implemented a tool, called PracExtractor, that can extract 338 recommendations and generate 173 specifications with reasonable accuracy from twelve large software manuals, including one for a large commercial software system. Additionally, with the generated specifications, PracExtractor have detected 1423 violations from 853 container images on DockerHub. We reported 325 of them and so far have got 47 confirmed as real configuration issues by the image maintainers from different organizations.

Interestingly, in addition to detecting system admins' configuration problems, PracExtractor can also help detect incorrect default settings for configuration parameters. When a default setting differs from a recommendation in the manual, it may indicate that the default setting is wrong. Incorrect default settings can easily cause configuration errors because system admins are most likely to go with default [60]. In our experiments, we did discover a few such software bugs, and three of them have already been confirmed respectively by MySQL and Cassandra.

PracExtractor is far from perfect. First, there is still much space to further improve its accuracy based on our analysis of false positives and false negatives (cf. Table 15). Further semantic analysis of parameter descriptions can improve the identification accuracy of parameter type, name and value. Second, PracExtractor currently cannot extract specifications with descriptive conditions, such as "set A with a large workload". This may possibly be handled by further incorporating domain knowledge of common descriptions and sub-clause analysis techniques.

Several other directions are also valuable to explore in the future. First, from our experience, we found that a uniform structure (e.g. per-parameter section) and consistent word usage (e.g. recommend) benefit extracting recommendations a lot. Therefore, it would be interesting to explore how manuals may be restructured so more information can be automatically extracted. Second, PracExtractor may be potentially used to detect documentation drift — manuals are not updated along with source code. By combining PracExtractor with source code analysis tools, it is possible to compare the configurations described in manuals and used in source code. Third, while PracExtractor focuses on analyzing user manuals, the approach may be applicable to extract good practices from other text-based documents such as knowledge-base (KB) articles, which are used by support engineers to troubleshoot customer issues.

Acknowledgments

We greatly appreciate the anonymous reviewers and our shepherd, Scott D. Stoller, for their insightful comments and feedback. We thank the Opera group, the Systems and Networking group at UCSD as well as Anita Jinda, Sam Lin and Jianmo Ni for useful discussions and paper proofreading. We thank Tianyin Xu for sharing a configuration issue dataset with us. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment.

References

[1] Cassandra release note. https://gitbox.apache.org/repos/asf?p=cassandra.git;a=blob;f=NEWS.txt;h=ead28f0ac3d8d93c1ad87a3b944c0c72345257c1;hb=HEAD.

[2] Configuration datasets. https://github.com/tianyin/configuration_datasets.

[3] Github issue. https://github.com/eviles/docker/issues/1.

[4] Github issue. https://github.com/sspreitzer/docker-httpd-mirror/issues/1.

[5] Github issue. https://github.com/newnius/Dockerfiles/issues/4/.

[6] Github issue. https://github.com/F21/hbase/issues/2.

[7] Github issue. https://github.com/binhnv/docker-hbase/issues/1.

[8] Github issue. https://github.com/Boostport/hbase-phoenix-all-in-one/issues/1.

[9] Github issue. https://github.com/oscerd/cassandra-image/issues/1.

[10] Github issue. https://github.com/femiwiki/cassandra/issues/3.

[11] Github issue. https://github.com/Babbleshack/docker-hadoop-yarn/issues/1.

[12] Github issue. https://github.com/jamesmcclain/HadoopDocker/issues/8.

[13] Github issue. https://github.com/vitessio/vitess/issues/5056.

[14] Github issue. https://github.com/madflojo/cassandra-dockerfile/issues/1.

[15] Github issue. https://github.com/publicisworldwide/docker-stacks/issues/31.

[16] Mysql bugzilla. https://bugs.mysql.com/bug.php?id=94487.

[17] Bhavish Agarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N Padmanabhan, and Geoffrey M Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, volume 9, pages 349–364, 2009.

[18] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, 2012.

[19] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, volume 10, pages 1–14, 2010.

[20] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[21] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International journal of human-computer interaction*, 17(3):333–356, 2004.

[22] CNN. Here's why you may have had internet problems today. https://money.cnn.com/2017/11/06/technology/business/internet-outage-comcast-level-3/index.html, 2017.

[23] Marie-Catherine De Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D Manning. Universal stanford dependencies: A cross-linguistic typology. In *LREC*, volume 14, pages 4585–92, 2014.

[24] Inc. Docker. Docker hub. https://hub.docker.com/, 2019.

[25] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56. USENIX Association, 2005.

[26] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16. ACM, 2016.

[27] Peng Huang. *Toward Understanding and Dealing with Failures in Cloud-Scale Systems*. PhD thesis, UC San Diego, 2016.

[28] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, 2018.

[29] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs.

[30] Yu Jin, Nick Duffield, Alexandre Gerber, Patrick Haffner, Subhabrata Sen, and Zhi-Li Zhang. Nevermind, the problem is already fixed: proactively detecting and troubleshooting customer dsl problems. In *Proceedings of the 6th International Conference on emerging Networking EXperiments and Technologies*, page 7. ACM, 2010.

[31] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 28–35. IEEE, 2004.

[32] Ben Maurer. Fail at scale: Reliability in the face of rapid change. *ACM Queue*, 13(8):30, 2015.

[33] Valerie Mendoza and David G Novick. Usability over time. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 151–158. ACM, 2005.

[34] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407. ACM, 2018.

[35] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151. ACM, 2014.

[36] David G Novick and Karen Ward. Why don't people read the manual? In *Proceedings of the 24th annual ACM international conference on Design of communication*, pages 11–18. ACM, 2006.

[37] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on internet technologies and systems*, volume 67. Seattle, WA, 2003.

[38] OWASP. Top 10-2017 a6-security misconfiguration. `https://www.owasp.org/index.php/Top_10-2017_A6-Security_Misconfiguration`, 2017.

[39] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 815–825. IEEE Press, 2012.

[40] C Perrow. Normal accidents: living with high-risk technologies (basic, new york). 1984.

[41] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. Confseer: leveraging customer support knowledge bases for automated misconfiguration detection. *Proceedings of the VLDB Endowment*, 8(12):1828–1839, 2015.

[42] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 451–466, 2016.

[43] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 193–202. IEEE Computer Society, 2011.

[44] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 131–140. IEEE, 2011.

[45] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE software*, 30(4):88–94, 2013.

[46] James Reason. *Human error*. Cambridge university press, 1990.

[47] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):64, 2017.

[48] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. Probabilistic automated language learning for configuration files. In *International Conference on Computer Aided Verification*, pages 80–87. Springer, 2016.

[49] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[50] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20. IEEE, 2011.

[51] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic

configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343. ACM, 2015.

[52] TechCrunch. Facebook blames a server configuration change for yesterday's outage. https://shorturl.at/opuEG, 2019.

[53] virtualizationreview. Configuration error leads to another amazon web services data breach. https://virtualizationreview.com/articles/2017/06/21/configuration-error-leads-to-another-aws-data-breach.aspx, 2017.

[54] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, volume 4, pages 245–257, 2004.

[55] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 53, pages 154–168. ACM, 2018.

[56] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143–164, 2004.

[57] Andrew Whitaker, Richard S Cox, and Steven D Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, volume 4, pages 6–6, 2004.

[58] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 620–631. IEEE Press, 2015.

[59] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 113–129, 2019.

[60] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.

[61] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *OSDI*, pages 619–634, 2016.

[62] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. How do system administrators resolve access-denied issues in the real world? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 348–361. ACM, 2017.

[63] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

[64] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 159–172. ACM, 2011.

[65] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 375–388. ACM, 2006.

[66] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 28–28. USENIX Association, 2011.

[67] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for java api functions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 380–391. IEEE, 2016.

[68] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 42(1):687–700, 2014.

[69] Sai Zhang and Michael D Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 312–321. IEEE Press, 2013.

[70] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE Computer Society, 2009.

[71] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, pages 27–37. IEEE Press, 2017.

Reverse Debugging of Kernel Failures in Deployed Systems

Xinyang Ge
Microsoft Research

Ben Niu
Microsoft

Weidong Cui
Microsoft Research

Abstract

Post-mortem diagnosis of kernel failures is crucial for operating system vendors because kernel failures impact the reliability and security of the whole system. However, debugging kernel failures in deployed systems remains a challenge because developers have to speculate the conditions leading to the failure based on limited information such as memory dumps. In this paper, we present Kernel REPT, the first practical reverse debugging solution for kernel failures that is highly efficient, imposes small memory footprint and requires no extra software layer. To meet this goal, Kernel REPT employs efficient hardware tracing to record the kernel's control flow on each processor, recognizes the control flow of each software thread based on the context switch history, and recovers its data flow by emulating machine instructions and hardware events such as interrupts and exceptions. We design, implement, and deploy Kernel REPT on Microsoft Windows. We show that developers can use Kernel REPT to do interactive reverse debugging and find the root cause of real-world kernel failures. Kernel REPT also enables automatic root-cause analysis for certain kernel failures that were hard to debug even manually. Furthermore, Kernel REPT can proactively identify kernel bugs by checking the reconstructed execution history against a set of predetermined invariants.

1 Introduction

Post-mortem diagnosis of software failures in deployed systems is becoming increasingly important for today's software development process. Many software vendors such as Microsoft and Apple have insider programs to test their latest software before it is released to the general public. Software developers rely more and more on debugging failures reported from early adopters to fix critical issues before every software release. In particular, the operating system is of the utmost importance because it is the foundation of the software stack and its bugs can have catastrophic impact on the reliability and security of the whole system.

Debugging kernel failures in deployed systems has been a challenge. The fundamental reason is that developers have to speculate on the conditions leading up to the failure based on the limited information available for post-mortem diagnosis such as crashing stacks or memory dumps. The complexity of the kernel makes developers' speculation ineffective in many cases. For example, the Windows kernel checks a set of invariants upon returning to the user mode, and terminates the system if any invariant is violated. Such a failure leaves developers an empty call stack, which makes it almost impossible to debug.

This motivates us to build a *practical* solution that enables developers to go back in time and examine the root cause of kernel failures in deployed systems. Reverse debugging is not a new idea [5, 10], and researchers and practitioners have developed record and replay solutions that can precisely log the execution of the whole system [16, 21, 22, 28]. However, existing whole-system record and replay solutions require the target operating system to run on top of emulation or virtualization, use an excessive amount of memory and storage space to support the record and replay, and introduce significant performance slowdown. On contrary, a practical reverse debugging solution for deployed systems must be able to provide a high-fidelity execution history for post-mortem diagnosis while meeting the requirements of low performance overhead, small memory footprint, no additional setup of software emulation or virtualization, minimal change to the operating system, and zero compromise on the backward compatibility with existing applications.

In this paper, we present Kernel REPT, the first practical solution for reverse debugging of kernel failures in deployed systems. It is an extension of REPT [19], a reverse debugging solution for user-mode applications. Kernel REPT leverages online hardware tracing to log the control flow of kernel executions and performs an offline binary analysis to recover the data flow. By configuring the hardware to trace the target kernel inside the kernel itself, Kernel REPT avoids the extra layer of software emulation or virtualization, has the minimal change to the target operating system, and is fully compatible with existing applications. Furthermore, hardware tracing is shown to be efficient [27].

Kernel REPT traces the kernel execution on each CPU core instead of on each software thread as done in REPT. This helps Kernel REPT achieve a small memory footprint because the number of CPU cores is much less than the number of software threads in a system. In this tracing configuration, one trace buffer may contain traces of multiple threads and the trace of one thread may span multiple trace buffers. To allow reverse debugging over the execution of a thread, Kernel REPT requires the context switch history to assemble the trace of the thread. However, Kernel REPT cannot infer the context switch history based on the control flow or the memory dump. Instead, Kernel REPT logs the context switch events during runtime and uses them to reconstruct the execution of a given thread during offline analysis. This way Kernel REPT can provide reverse debugging over the execution of a thread on top of the core-based tracing.

REPT performs forward and backward instruction emulation to recover the program's data flow, however, it is insufficient to just emulate the semantics of machine instructions in Kernel REPT. This is because a processor modifies the kernel state when a hardware event such as interrupts or exceptions occurs. To correctly recover the kernel state, Kernel REPT needs to emulate the semantics of these hardware events properly. However, these hardware events are not explicitly logged in the control flow trace, and different events may make different changes to the kernel state. Kernel REPT solves this problem by leveraging the kernel configuration of hardware event handlers. For instance, Kernel REPT can tell a page fault just happened when the page fault handler is executed as shown in the control flow trace.

We implement Kernel REPT and deploy it on a billion devices running Microsoft Windows. Our experiments show that Kernel REPT is efficient as it incurs less than 10% slowdown for microbenchmarks and 2% slowdown for applications like Nginx and Chrome. Windows kernel developers use Kernel REPT to debug real-world kernel failures and find the root cause of some kernel bugs that have existed for a decade and caused innumerable failures.

The usage of Kernel REPT is not limited to interactive reverse debugging. To this end, we develop an automatic root-cause analysis for a common class of kernel failures where the kernel fails when it detects that certain resources are not properly released before returning to the user mode. This class of failures is hard to debug even manually without Kernel REPT because the kernel stack is empty when a failure happens. Based on the execution history reconstructed by Kernel REPT, our analysis can *automatically* identify the buggy function for 136 out of 188 real-world kernel failures of this class. This automatic root-cause analysis is deployed as part of Microsoft's error reporting service [24].

The reconstructed kernel execution history can enable not only automatic root-cause analysis but also proactive bug detection. A common kernel bug pattern is that the *exception* handling code fails to properly release resources acquired during the execution wrapped in the *try* block. We build a hybrid analysis to proactively look for this bug pattern by dynamically analyzing the execution in a *try* block and statically analyzing the code in the *exception* handling block. By analyzing thousands of real-world kernel execution histories, our hybrid analysis finds 17 new bugs in the Windows kernel, and all of them are confirmed and fixed.

2 Overview

The goal of Kernel REPT is to reconstruct the execution history of kernel failures in deployed systems for effective post-mortem diagnosis without incurring noticeable runtime overhead. As an extension of REPT [19], it utilizes hardware tracing to log the kernel's control flow to a circular buffer at runtime, and recovers its data flow by running binary analysis on the recorded control flow and the memory dump of a failure. In the rest of this section, we first provide a background of REPT. Then we discuss the challenges faced by Kernel REPT.

2.1 REPT

REPT shows a promising way to do reverse debugging for user-mode failures in deployed systems. REPT logs a program's control flow into a *per-thread* circular buffer with low runtime overhead via hardware tracing (e.g., Intel Processor Trace [18]), and then recovers its data flow offline by combining the control flow with the memory dump taken at the failure point. To do so, REPT runs an iterative binary analysis that performs forward and backward instruction emulation on the recorded instruction sequence to infer the program state before every instruction based on the final state stored in the memory dump. REPT checks for conflicts during the execution history reconstruction and performs error corrections based on heuristics. REPT also supports multi-threaded programs by merging the instruction sequences of different threads into a partially ordered single instruction sequence based on the fine-grained timestamps logged by the hardware tracing, and limiting the use of concurrent shared memory writes in the binary analysis if their exact order cannot be determined. The reconstructed execution history is not perfect, but it is shown that REPT achieves high accuracy and enables effective reverse debugging of real-world application failures.

To illustrate how REPT works, we show an example borrowed from the REPT paper [19, Figure 2] in Figure 1. This example has 5 instructions in the control-flow trace ($I_1..I_5$). The program state S_i represents the state *after* the execution of instruction I_i. Therefore, the final program state S_5 stored in the memory dump has `rax`=3, `rbx`=0, and `[g]`=3. REPT performs backward and forward analysis iteratively

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑ {rax=?, rbx=?, [g]=3} →		↑ {rax=?, rbx=?, [g]=2}
I_1	`lea rbx, [g]`	S_1	↑ {rax=?, rbx=?, [g]=3}	↓ {rax=?, rbx=g, [g]=3}	↑ {rax=?, rbx=g, [g]=2}
I_2	`mov rax, 1`	S_2	↑ {rax=?, rbx=?, [g]=3}	↓ {rax=1, rbx=g, [g]=3}	↑ {rax=1, rbx=g, **[g]=2**}
I_3	`add rax, [rbx]`	S_3	↑ {rax=3, rbx=?, **[g]=3**}	↓ {**rax=3**, rbx=g, [g]=3}	↑ {rax=3, rbx=g, **[g]=?**}
I_4	`mov [rbx], rax`	S_4	↑ {rax=3, rbx=?, [g]=3}	↓ {rax=3, rbx=g, [g]=3}	↑ {rax=3, rbx=g, [g]=3}
I_5	`xor rbx, rbx`	S_5	↑ {rax=3, rbx=0, [g]=3}	↓ {rax=3, rbx=0, [g]=3} →	

Figure 1: "This example shows how REPT's iterative analysis recovers register and memory values when there exist irreversible instructions with memory accesses. We use "?" to represent "unknown", and use "g" to represent the memory address of a global variable. Some values are in bold-face because they represent key updates in the analysis. We skip the fourth iteration which will recover [g]'s value to be 2 due to the space constraint." [19, Figure 2]

to recover data values. In the first iteration, REPT does not update the global variable `[g]` in S_3 because `rbx`'s value is unknown. In the second iteration, there is a conflict for `rax`'s value in S_3. Its original value is 3, but the forward analysis would infer value 4 for it (`rax + [g] =` 1 + 3 = 4). REPT keeps the original value of 3 because it is from the final program state stored in the memory dump. In the third iteration, REPT recovers `[g]`'s value to be 2 based on `rax`'s value before and after the `add` instruction I_3.

For the purpose of this paper, we abstract REPT as a mechanism that takes as the input a final machine state and its preceding instruction sequence, and outputs the recovered machine state before every instruction with high accuracy. Kernel REPT leverages this data recovery mechanism to enable reverse debugging of kernel failures.

2.2 Challenges

A straw-man solution to support reverse debugging of kernel failures is to modify REPT to trace the kernel execution of each software thread and run the same binary analysis on a kernel memory dump. However, this simple solution does not work for two reasons.

First, it incurs unacceptable memory overhead. The kernel is shared by all threads on a system and allocating a trace buffer for each of them can consume an unpredictable amount of memory, especially when a system can have thousands or even tens of thousands of threads.

Second, the kernel has to handle hardware events of which user-mode applications are unaware. For instance, interrupts and exceptions can change the kernel's stack layout without executing any explicit instruction. The details of these hardware events such as the exception vector are not logged by the hardware tracing. However, such information is important for the data flow recovery because different types of hardware events have different architectural effects that must be emulated.

3 Kernel REPT

In this section, we present the design of Kernel REPT. We first describe how Kernel REPT avoids excessive memory consumption via per-core tracing while still allowing reverse debugging over the execution of a thread. Then we present how Kernel REPT handles hardware events when performing the offline binary analysis to recover the data flow.

3.1 Per-Core Tracing

To minimize the memory footprint, Kernel REPT chooses to do *per-core tracing* instead of per-thread tracing for the kernel. That is, Kernel REPT allocates a circular trace buffer for each logical core and logs the kernel-mode execution on a core to its corresponding buffer. Kernel REPT does not log the control flow of user-mode executions because the actual machine code executed in the user mode is not directly related to the root cause of kernel failures. Per-core tracing ensures that Kernel REPT's memory usage is linear in the number of logical cores on a system, and the number of logical cores is fixed and small compared to the number of software threads. This allows Kernel REPT to configure a large trace buffer for each core when necessary without the risk of exhausting the memory.

Per-core tracing does come with its own problems. It is more intuitive for developers to follow the execution on a software thread as opposed to a hardware core. On a multiprocessor system, per-core tracing may mix traces of different threads into one trace buffer and spread the trace of a single thread into multiple trace buffers. This requires Kernel REPT to obtain the context switch history to identify the trace of a single thread.

Ideally, Kernel REPT should recover the context switch history from a per-core trace by leveraging the binary analysis. However, the binary analysis cannot effectively reverse the context switch routine because the scheduling history is neither preserved in the memory dump nor can be inferred from the recorded instruction sequence. We show the pseudo code of a typical context switch routine in Figure 2. The context switch routine saves the register context of the previous

```
; pseudo code for context switch
; rdi points to the old thread
; rsi points to the new thread
push rax                  ; save GPRs
push rbx
...
mov KernelStack[rdi], rsp ; switch stack
mov rsp, KernelStack[rsi]
...                       ; restore GPRs
pop rbx
pop rax
ret
```

Figure 2: Pseudo code for context switch. Basically, the context switch routine saves the register context to the previous thread's stack and the stack pointer to the previous thread's internal data structure, and then restores the context of the new thread by doing the opposite operations.

Type	Origin	Details
Interrupt	User/Kernel	Vector number
Exception	User/Kernel	Vector number
Syscall	User	N/A

Table 1: Information about hardware events needed for software emulation.

thread on its stack, swaps the stack pointer, and then restores the register context of the newly scheduled thread to resume its execution. In this process, the context switch routine does not save the information about the previous thread before resuming the execution of the new thread, and hence the binary analysis is unable to recover the scheduling history. This also makes the binary analysis ineffective when applied directly to the per-core trace because the register values before a context switch cannot be recovered. Therefore, Kernel REPT chooses to log the context switch history in software.

3.2 Handling Hardware Events

The operating system manages hardware resources and has to handle hardware events. Therefore, the architectural effects of these hardware events, which are transparent to user-mode execution, are part of the kernel-mode execution and must be emulated when running the binary analysis for kernel data recovery.

Different hardware events have different architectural effects, and Kernel REPT has to understand the semantics of each hardware event for emulation. We list the information about the hardware events required for software emulation in Table 1. Specifically, Kernel REPT has to not only tell the type of a hardware event, but also infer whether this event occurred in the user or kernel mode and what it was about.

To do so, Kernel REPT first infers the occurrence of a hardware event based on the hardware trace. Given that Kernel REPT only traces the kernel-mode execution, hardware tracing will be paused when the execution returns to the user mode, and resumed when the execution enters the kernel mode. Therefore, the signal of tracing being resumed already indicates the occurrence of a hardware event—a system call or an interrupt/exception happening in the user mode. The hardware trace logs the occurrence of an asynchronous event during the kernel execution. Kernel REPT uses such information to detect the occurrence of an interrupt or exception in the kernel mode.

Next, Kernel REPT needs to infer additional information about a hardware event such as its type and the vector number for an interrupt. The Windows kernel configures the handlers for these hardware events at the boot time and never reconfigures the settings throughout the rest of its lifetime. Kernel REPT assumes this invariant holds for the vast majority of kernel failures under non-adversarial scenarios, and determines the event type as well as the vector number for interrupts/exceptions by comparing the control flow to the kernel configuration stored in the memory dump.

Finally, Kernel REPT emulates the architectural effect of these events according to the hardware specification. Kernel REPT performs the emulation during the binary analysis as if it were emulating a special instruction. However, not all data values are available to Kernel REPT when emulating a hardware event. For example, when emulating an exception from the user mode, Kernel REPT does not know the user-mode instruction that triggers the exception, so it cannot fill up all fields of the trap frame. Similarly, Kernel REPT does not log the parameters of system calls, so it does not necessarily have the register context of a system call event if it cannot be recovered from the memory dump. In these cases, Kernel REPT simply marks the register and memory values as unknown to avoid propagating stale values during the binary analysis.

4 Automatic Analyses

In this section, we present two automatic analyses enabled by Kernel REPT. The first analysis is an automatic root-cause analysis that can identify the buggy function for a specific class of kernel failures. The second analysis is a hybrid analysis that can *proactively* detect bugs that may lead to this class of kernel failures.

4.1 Root-Cause Analysis

A common kernel bug is that calls to *do* operations (e.g., resource acquisition) are *not* matched by calls to *undo* operations (e.g., resource release). For example, if the kernel disables interrupts before entering a critical region but fails to re-enable interrupts after leaving the critical region, the system will crash eventually. Despite the simple nature of this failure type, it is difficult to debug kernel failures caused by these bugs simply based on a memory dump. The key challenge is that the buggy function that missed the *undo* operation may have returned a long time ago. Without an execution history, it is hard to infer which function could be the buggy one. Particularly, the Windows kernel checks if there is a missing *undo* operation (e.g., resource not released) before it returns to the user mode. A failed check leaves developers an empty call stack, which makes it almost impossible to debug. What makes the matter worse is that some operations allow recursion by maintaining a counter for all pending *do* operations (e.g., recursive lock). This requires the developers to match the *do* and *undo* operations in a potentially long history before they can identify the unmatched *do* operation, which further complicates the diagnosis process.

The root-cause analysis identifies the buggy function that misses the *undo* operation by searching along the execution history to find the *first* function where a specified value changes between the function entry and return. For example, to detect when the kernel fails to re-enable the interrupts upon exiting a critical section, the analysis checks for the interrupt enablement at each function entry and return, and reports the first one that has a mismatched value. However, there are exceptions to the above analysis because some functions are designed to just perform the *do* or *undo* operation. For example, if enabling/disabling interrupts is implemented in a library function, then the function is expected to modify the value between its entry and return. The library functions that are designed to perform only a *do* or *undo* operation are relatively stable across kernel versions, so we maintain a whitelist for such functions. The root-cause analysis ignores them when searching for the buggy function.

4.2 Proactive Bug Detection

A common bug pattern that causes *undo* operations being missed is related to the try/catch-like primitives designed to handle hardware exceptions gracefully. For example, the Windows kernel uses Structured Exception Handling (SEH) [9] to handle page faults when accessing a user-mode page. An *undo* operation may be missed when an exception occurs if the try scope contains a *do* operation and the catch scope does not have the corresponding *undo* operation. We show an example of this bug pattern in Figure 3. foo calls read_user_obj in a try block to handle page faults in case the user-mode page is not mapped with proper permissions (line 12). read_user_obj temporarily disables Supervisor-Mode Access Prevention (SMAP) in order to access user-mode pages (line 4). If a page fault occurs when read_user_obj dereferences user_ptr, the page fault handler will redirect the execution back to the catch block in foo (line 15), skipping the subsequent call to enable_smap (line 5). The correct implementation is to apply the scope of the try block to the dereference of user_ptr instead of the entire read_user_obj function.

```
1  obj_t read_user_obj(int *user_ptr) {
2      obj_t obj;
3      disable_smap();
4      obj.a = *user_ptr;
5      enable_smap();
6      return obj;
7  }
8
9  int foo() {
10     obj_t obj;
11     try {
12         obj = read_user_obj(user_ptr);
13     }
14     catch {
15         return -1;
16     }
17     return 0;
18 }
```

Figure 3: Example code that misuses try/catch leading to a missing *undo* operation.

Leveraging the execution history reconstructed by Kernel REPT, we design an automatic *hybrid* analysis to proactively detect bugs of this pattern. Our hybrid analysis has two steps. First, it uses a dynamic analysis to check if there is any *do* operation in a try scope based on the execution history. Second, it uses a static analysis to check if the matching catch scope does *not* have the corresponding *undo* operation. The analysis identifies try and catch scopes based on the unwind information in the binaries [14].

The assumption behind this analysis is that a hardware exception may happen at any time within the try scope, and missing the *undo* operation in the catch scope means that the kernel would fail to restore the state if an exception happens after the *do* operation. Even though this assumption may not be true for all cases, a violation implies an overuse of the try scope that should be addressed by developers.

The hybrid analysis is accurate and effective because it leverages execution traces from a huge number of deployed systems. First, a try scope may include a significant amount of execution involving multiple levels of function calls. Statically analyzing such a try scope is challenging, and our dynamic analysis avoids this challenge. Second, the executions of all kernel threads (i.e., not limited to the failure thread) in a failure report are used in the hybrid analysis. This allows the analysis to avoid the common constraint on completeness for dynamic analysis. Third, the code logic in the catch scope is usually straightforward, so simple static analysis is sufficient for checking if an expected *undo* operation exists.

5 Implementation

In this section, we describe the implementation and deployment of Kernel REPT on Microsoft Windows.

5.1 Kernel Tracing

Kernel REPT logs both the context switch history and the control flow of the Windows kernel. To log the context switch history, Kernel REPT leverages Event Tracing for Windows (ETW) [4]. ETW logs the timestamp, identifiers of both the old thread and the new thread for each context switch event. These ETW events will be included in the memory dump of a kernel failure.

To record the kernel's control flow, Kernel REPT enables Intel Processor Trace (PT) [18] on each processor core for the kernel-mode execution at system start. We adapt the driver from REPT to enable Intel PT for the Windows kernel. Our driver change has roughly 2K lines of C code. We mark the virtual memory of trace buffers as read-only to prevent the Windows kernel from accidentally corrupting them. This can be done because Intel PT outputs the trace directly to the physical memory and is not subject to the page permission we set on the virtual memory. Similar to the user-mode tracing in REPT, the kernel-mode trace is stored in the memory dump when a kernel failure is reported.

Kernel REPT currently disables multithreaded analysis for kernel failures due to a caveat of Intel PT. The timestamp logging of Intel PT cannot be configured for a specific privilege level. Without such a privilege-level filtering, the timestamps generated during the user-mode execution will overwrite the kernel's control-flow trace in the circular buffer. One possible solution is to dynamically enable and disable timestamps in software when the processor switches between the user and kernel mode. We leave its exploration to future work.

5.2 Trace Parsing

Intel PT encodes the control flow in a highly compact format. It requires the code binary to parse the trace to reconstruct the control flow. Meanwhile, an operating system can swap out kernel code pages to reclaim its underlying physical memory. This can fail the trace parsing because even capturing the entire physical memory upon a kernel failure can be insufficient due to the unavailability of swapped-out code pages. One possible solution to this problem is to lock all the kernel code pages into the physical memory, but this will increase the memory pressure to the overall system.

In Kernel REPT, we choose to reconstruct the code pages based on the image's metadata stored in the memory dump and its binary file. One disadvantage of this approach is that it does not work for third-party drivers where the binary files are unavailable. This is not a big issue in practice because the code that was executed close to the failure point is usually available in the physical memory (thanks to the memory manager's policy).

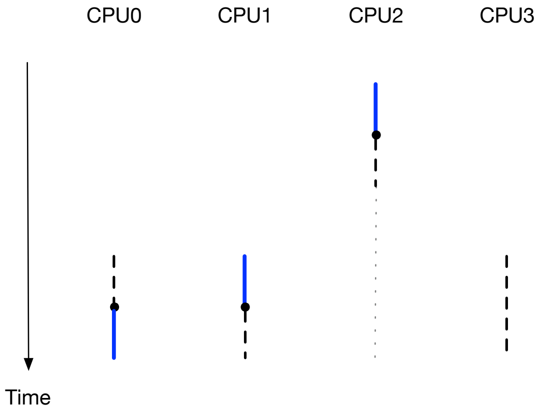


Figure 4: An example scenario where traces of a thread are not contiguous. Solid lines represent the execution trace of the interesting thread. Dashed lines represent the execution trace of other threads. Dots (•) connecting them represent context switches. Dotted lines mean the processor was in sleep mode and no execution trace was generated.

5.3 Binary Analysis

We implement Kernel REPT's binary analysis in 15K lines of C++ code on top of REPT. It includes the emulation of hardware events, the thread trace reconstruction based on the context switch history, and the two automated analyses. Most of the implementation is straightforward, but there are two technical details worth mentioning here.

First, a thread's trace may be *noncontiguous* with respect to the thread's execution. We show an example in Figure 4. In this example, core 2 was in sleep mode for a long period of time, and no execution trace was generated. The trace of the target thread in its circular buffer can be obsolete and disconnected from the rest of the thread's trace on other cores. This can happen when the trace of the target thread on core 1 was overwritten by another thread, and the overwritten trace was more recent than the trace in core 2's trace buffer. Kernel REPT checks for such cases based on the timestamps of context switch events, and discards those disconnected traces.

Second, certain kernel instructions can be missing from the control-flow trace when the system is running inside a virtual machine. In a virtualized environment, guest instructions that can modify the system state (e.g., `wrmsr`) are trapped and emulated by the hypervisor. Kernel REPT only traces the kernel-mode execution, so its binary analysis will be under the illusion that these instructions are skipped according to the Intel PT trace. This can result in an inconsistent state in the data flow recovery. Kernel REPT solves this issue by detecting VMEXIT and adding the skipped instruction back to the instruction sequence if it is deemed as emulated by the hypervisor. Specifically, for each asynchronous event logged in the trace, Kernel REPT checks if there is one and only one possible instruction between the current instruction and the next instruction in the instruction sequence. If so, Kernel REPT determines that the instruction is emulated by the hypervisor, and adds it back to the instruction sequence before running the binary analysis. This check is straightforward to implement because the hypervisor-emulated instruction is typically a non-branch instruction.

5.4 Deployment

We have deployed Kernel REPT in the ecosystem of Microsoft Windows. The deployment of Kernel REPT spans three parts: Windows, Windows Error Reporting (WER) [24], and Windows Debugger [12].

On Windows, we released the kernel driver that configures Intel PT tracing and the ETW context switch logger for the kernel, and a user-mode daemon that communicates with WER to decide when to start/stop the driver. These components were released as part of Windows 10 version 1803.

On the WER service, we added support for requesting Intel PT traces for a given kernel failure. When the WER service receives such a request, it selects devices that have reported the same kernel failures in the past and are capable of Intel PT to enable tracing for future failure reporting. The WER service also runs the automatic root-cause analysis on kernel failures of the specific error code [2].

On Windows Debugger, we implemented Kernel REPT's interactive reverse debugging by extending REPT's debugger extension. This extension allows an developer to set breakpoints, go back and forth on the reconstructed execution history, switch to different threads, and inspect the local and global variables.

6 Evaluation

In this section, we evaluate the performance and effectiveness of Kernel REPT. For performance, we run both microbenchmarks and real-world applications with Kernel REPT enabled to measure the runtime overhead. For effectiveness, we evaluate Kernel REPT's data recovery and report how it helps developers debug real-world kernel bugs.

6.1 Performance

We evaluate the performance impact of Intel PT tracing and context switch logging on kernel-mode execution by running UnixBench 5.1.3 [11], ApacheBench [1] on Nginx 1.17.5 [7], and JetStream 2 benchmarks [6] on Chrome 79 [3]. We choose UnixBench because it measures the micro-level performance impact on a kernel's key functions such as system calls and context switches. We choose Nginx and Chrome because they represent popular programs for server and client scenarios, respectively. We run the experiments on a Windows 10 (version 1903) machine equipped with an Intel i7-6700K processor (8 logical cores) and 16GB RAM. We allocate two separate circular buffers for each logical core: a 1MB buffer for Intel PT tracing and a 128KB buffer for context switch logging. We choose this default setting empirically for experiments, but our real-world deployment allows developers to adjust the configuration as needed.

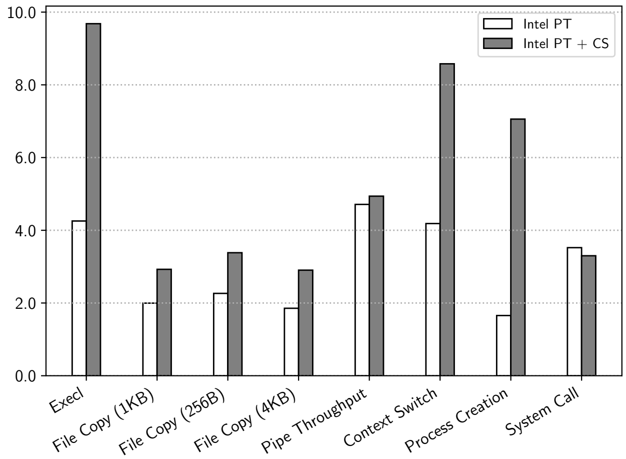


Figure 5: Performance overhead of running UnixBench with Intel PT tracing and context switch logging.

6.1.1 UnixBench

We run UnixBench on the Windows Subsystem for Linux 1 (WSL 1) [13]. WSL 1 implements Linux system calls from the Windows kernel to run unmodified Linux ELF binaries such as UnixBench. We show the performance results of UnixBench in Figure 5. For Intel PT tracing only, the average performance overhead is 3.06% with no single benchmark exceeding 5% overhead. With the context switch logging enabled in addition, the average performance overhead becomes 5.35% with the highest performance overhead of 9.68%. In particular, three benchmarks, Execl, Context Switch and Process Creation, have more frequent context switches than other benchmarks. Therefore, they have higher overhead when context switch logging is turned on.

6.1.2 Nginx

We evaluate the performance overhead of different tracing setups on Nginx using ApacheBench. We run Nginx on the test machine with 8 logical cores and 16GB RAM. We use the default configuration provided by Nginx but modify worker_processes to 8 to use all the logical cores. We

	# Instructions	Data Recovery
IRQL Fault (Kernel)	3,310,906	65.13%
Code Overwrite	1,151,315	73.18%
Stack Trash	315,046	65.75%
IRQL Fault (User)	9,176,219	61.56%
Stack Overflow	10,421,430	60.97%
Hardcoded Breakpoint	9,129,048	61.65%
Double Free	5,232,343	43.03%

Table 2: Kernel REPT's data recovery rate on kernel failures caused by notmyfault [8].

run ApacheBench on a separate client machine with 16 logical cores. We use the client to make 100,000 HTTP requests over 16 concurrent connections and then measure the throughput (requests/second). We run each session 10 times and report the average throughput. The reduction of the average throughout when the Intel PT tracing is enabled (with or without the context switch logging) is about 2%.

6.1.3 Chrome

We run JetStream 2 benchmarks on the Chrome browser to evaluate the performance impact on web browsing, one of the most common client-side scenarios. There is no visible performance slowdown when both the Intel PT tracing and the context switch logging are enabled. We believe this is because the benchmarks have most of their computation in the user mode.

6.2 Effectiveness

We evaluate the effectiveness of Kernel REPT from four perspectives: (1) how well it can recover data; (2) how its interactive reverse debugging can help developers debug kernel bugs; (3) how accurate the automatic root-cause analysis is; (4) how well the proactive bug detection works.

6.2.1 Data Recovery

We evaluate Kernel REPT's data recovery based on the same metric as REPT [19]. Specifically, we measure the data recovery rate as the percentage of *register uses* (i.e., a register used as the source operand or in the address of a memory operand) for which the register value is recovered by Kernel REPT. Register use is a good metric because it avoids double counting (e.g., we only count it once when rax and rbx are both known in an instruction mov rax,rbx) and memory values are often loaded into registers first before being used in an operation.

In our experiment, we trigger Windows kernel failures by using a public utility program called notmyfault [8], which injects a kernel driver to cause various types of failures such as stack overflow and double free. We pick notmyfault because its injected failures are reproducible. Our experiment does not include the Buffer Overflow fault in notmyfault because it cannot trigger a kernel failure on the latest Windows. For each failure, we count the number instructions and measure the data recovery rate for the crashing thread. Our experimental results are shown in Table 2.

```
bool get_desc(..., desc_t **p) {
    int size;
    bool success;
    *p = malloc(sizeof(desc_t));
    driver = find_driver();
    success = (driver->op)(*p, &size);
    return success;
}

void foo() {
    desc_t *p;
    bool success;
    if (...) {
        success = get_desc(..., &p));
    } else {
        success = get_desc(..., &p));
    }
    if (success) {
        bar(p->owner->sid); // CRASH!
    }
}
```

Figure 6: A real-world example that showcases how Kernel REPT helps developers find bugs.

Kernel REPT's data recovery rate is over 60% for all but one failure even when some execution contains over 10 million instructions. The Double Free case involves a series of memory allocation/free operations. As reported in REPT, memory allocation operations are hard to reverse because their metadata may be completely overwritten by subsequent free and reallocation operations. We believe this is the main reason for the Double Free case to have a lower data recovery rate than others.

Comparing with REPT's data recovery on user-mode programs [19, Figure 4], we can see that Kernel REPT achieves a similar data recovery rate for kernel failures. Note that some recovered data may be incorrect, but we cannot directly measure it due to the lack of a ground truth. However, we expect it to be in the same low percentage as REPT.

6.2.2 Interactive Reverse Debugging

We use a real-world case to demonstrate how Kernel REPT can help developers debug kernel bugs through interactive reverse debugging. This bug was introduced to the Windows kernel more than a decade ago. It was not fixed until Kernel REPT became available due to the lack of information in memory dumps. We show a simplified version of the code around the bug in Figure 6. In the code snippet, `foo` calls `get_desc` to receive a pointer to a descriptor object. Depending on certain conditions (line 13), the call can happen at two places with potentially different parameters (line 14 and 16). `get_desc` allocates the memory for the descriptor object (line 4) and finds the driver that provides the corresponding callback (line 5). Then, `get_desc` invokes the driver's callback function to initialize the object, which returns whether the initialization succeeds and the number of bytes being initialized (line 6). Finally, the crash happens when `foo` dereferences a pointer field (`owner`) inside the descriptor object (line 19).

To debug this kernel failure, a developer first has to determine where `get_desc` is called (line 14 or 16). Without Kernel REPT, a developer would need to use some auxiliary information to figure it out. However, with the recorded control flow, it is straightforward to find it. The next step is to determine the target function of the callback (line 6). This can be challenging without the recorded control flow because `get_desc` has already returned and the relevant information may no longer be available. In fact, the actual code involves multiple levels of indirect function calls, making the problem even harder. With Kernel REPT, the developer can easily reach the correct target function based on the recorded control flow. Finally, the developer has to understand how the descriptor is mis-initialized by the callback function. In this particular case, it turns out that the callback function does not attempt to initialize the descriptor object at all. It just returns success with the number of initialized bytes being zero. Unfortunately, `foo` does not check the number of bytes being initialized, leading to the subsequent crash caused by dereferencing an uninitialized pointer value. To fix this bug, the developer changes the callback function to return an error code indicating that the operation is not supported.

6.2.3 Root-Cause Analysis

We run the automatic root-cause analysis described in §4.1 on 377 real-world kernel failures of a specific error code [2] reported to Microsoft over two weeks. This error code is used by the kernel when it detects a specific resource is not properly released upon returning to the user mode. The analysis identifies potential buggy functions in 33 kernel components including the core OS kernel, the GUI subsystem, the file system, and some third-party drivers. To evaluate the accuracy of the root-cause analysis, we manually check each identified buggy function either based on the source code or the confirmation from developers. Since the source code of third-party drivers is unavailable and it is difficult to reach their developers, we exclude the 189 kernel failures whose buggy functions are in a third-party driver.

We show the accuracy of the root-cause analysis for the remaining 188 kernel failures in Table 3. The root-cause analysis correctly identifies the buggy function for 148 failures. 136 of these failures are caused by unsafe `try/catch` operations and 12 of them are caused by other miscellaneous issues (e.g., the code fails to properly clean up the state on an error handling path). The root-cause analysis fails to identify the true buggy function for 40 kernel failures. We manually analyze them via interactive reverse debugging and find that we can find the true buggy function for 23 failures. The rest 17 failures cannot be resolved due to the limited trace size or data recovery.

True Blame		False Blame	
Try/Catch	Misc.	Manual	Unresolved
136	12	23	17

Table 3: The accuracy of the automatic root-cause analysis on 188 real-world kernel failures for which Kernel REPT blames a function in the first-party components.

While analyzing the memory dumps of the 23 failures manually, we find that one common reason for the automatic root-cause analysis to miss the true buggy function is that the *do* and *undo* operation pair is tied to an object's lifetime instead of a function's lifetime. For example, one way to manage the acquisition and release of a lock is to implement the acquire operation in a constructor function and the release operation in the corresponding destructor function. In this case, a function can indirectly acquire the lock by creating such an object, and then passes it to another function that releases this lock by destructing the object. If the kernel fails to destruct the object due to programming errors, it not only causes memory leaks but also leads to the missing *undo* issue. The root cause of such programming errors can vary case by case, and blaming the function that creates the object and seemingly misses the destruction does not always lead to the correct outcome. However, even in these cases, the root-cause analysis can provide useful information to help developers find the root cause.

6.2.4 Proactive Bug Detection

We run the proactive bug detection described in §4.2 over 2000 execution histories reconstructed from memory dumps of real-world kernel failures. We do not limit the failure type, and use the execution histories of *all* threads in a memory dump. We use memory dumps of kernel failures instead of normal executions because they are currently the major source of recorded real-world kernel executions.

We have found a total of 17 previously unknown kernel bugs, and all are confirmed and fixed. For one of the bugs, we observed an actual kernel failure caused by it a few days after we reported it to the developer. This shows the potential of using Kernel REPT to uncover bugs even before they are triggered in practice.

7 Discussion

Kernel REPT extends REPT's support for reverse debugging of user-mode failures to kernel-mode failures. Therefore, it shares two limitations with REPT. First, the reconstructed execution history is incomplete because the circular trace buffer only captures a fixed amount of control-flow information before the kernel failure. Second, the reconstructed execution history is imperfect because many instructions are not reversible. Despite the two limitations, Kernel REPT's reverse debugging capability allows successful diagnosis of many real-world kernel failures that were impossible to debug before.

The automatic root-cause analysis described in §4.1 requires a whitelist of functions that perform only a *do* or *undo* operation by design. It requires manual effort to construct and update the whitelist. The root-cause analysis may have false blames due to the incompleteness of the whitelist. In practice, we rely on developers' feedback to keep the whitelist up to date.

One interesting observation we have is that REPT-style reverse debugging is more effective for kernel-mode failures than for user-mode failures. We believe the key reason is that the Windows kernel operates in a more *defensive* manner: it performs various checks of invariants in kernel state at different times of the execution, such as checking missing undo operations before returning to the user mode (see §4.1 for details). These checks shorten the execution between the program defect and the program failure. This is crucial for the effectiveness of REPT-style reverse debugging since its reconstructed execution history is incomplete and imperfect.

8 Related Work

In this section, we discuss the previous work related to Kernel REPT in three categories: record and replay, failure analysis and failure reproduction. We omit the discussion of REPT [19] as we have covered it comprehensively in §2.1.

8.1 Record and Replay

Record and replay tools have been applied to debugging for both user-mode applications [15, 25, 30, 33] and operating systems [16, 21–23, 28, 31, 32, 34, 35]. Software-based record and replay tools [16, 21–23, 28, 35] for operating systems require running the whole system in a virtualized environment to log all non-deterministic inputs to the target system. These tools are rarely deployed in production environments because of their significant runtime overhead and compatibility issue. The latter is caused by the requirement for a special setup such as installing a custom virtual machine.

Hardware-based record and replay tools [31, 32, 34] modify the underlying hardware to record the execution of a target system. For example, Flight Data Recorder (FDR) [34] instruments the processor's cache coherency protocol to enable record and replay of a multiprocessor system. The required hardware modification makes these systems expensive to build and adopt in practice.

Compared to the above record and replay tools, Kernel REPT enables effective reverse debugging for operating systems running on commodity hardware with low performance and space overhead at runtime, making it practical for deployment on real-world systems.

8.2 Failure Analysis

RETracer [20] is a triaging system for both user-mode and kernel-mode failures. It starts with a corrupted pointer, performs backward taint analysis on memory dumps, and assigns the blame to a function that contributes to the access violation. RETracer uses the crashing stack as an approximate execution trace when performing backward taint analysis, so it cannot effectively analyze kernel failures with an empty call stack.

Postmortem Symbolic Evaluation (PSE) [29] performs static backward slicing on memory dumps to identify where a bad pointer is originated. PSE is also limited by the information available in the dump, and can have false positives due to unresolved memory aliases.

SherLog [36] analyzes the log messages generated during a failed execution to infer control and data values before the failure point. While this approach may be useful to diagnose logical bugs in a program, log messages cannot be used to diagnose low-level software bugs such as memory safety errors. In addition, its effectiveness depends on the developer's expertise in determining the key information to log, which varies case by case.

Kernel REPT is complementary to the above production failure analysis techniques. For instance, we integrated REPT-style reverse debugging into RETracer so that the latter can run its backward taint analysis on the reconstructed execution history to derive a more precise blame for production failures.

8.3 Failure Reproduction

Execution Synthesis (ESD) [37] explores possible program paths to search for inputs that can lead to the same failure. ESD relies solely on memory dumps, and its symbolic execution [17] may not be able to solve complicated constraints when exploring a long execution history of complex program state. This makes it difficult to work for complex programs such as the operating system kernel.

BugRedux [26] reproduces a production failure by instrumenting the program to collect execution data at different levels and employing symbolic execution to compute an input leading to a similar execution. Program instrumentation incurs overhead even for normal executions, and symbolic execution is known to have path explosion problems.

Kernel REPT allows developers to examine the execution history of a kernel failure without the need to reproduce it.

9 Conclusion

We have presented the design and implementation of Kernel REPT, the first practical solution for reverse debugging of kernel failures in deployed systems. Kernel REPT records the kernel's control flow and context switch events on each processor, and recovers its data flow on each software thread via binary analysis. Its analysis emulates both machine instructions and hardware events such as interrupts and exceptions. In addition to the support for interactive reverse debugging, we have developed two automatic analyses on top of Kernel REPT: a root-cause analysis that can identify the buggy function for a class of kernel failures, and a hybrid analysis that can proactively detect bugs due to a misuse of the `try/catch` primitive. We show that Kernel REPT is efficient for real-world deployment and effective for debugging real-world kernel failures.

Acknowledgments

We thank our shepherd, David Devecsery, and other reviewers for their insightful feedback during this unprecedented time. We are also very grateful for all the help from our colleagues at Microsoft during the development and deployment of Kernel REPT. In particular, we thank Alan Auerbach, Joe Ballantyne, Harini Bandi, Abolade Gbadegesin, Stephen Hufnagel, Mehmet Iyigun, Pedro Justo, Scott Lee, Graham McIntyre, Timothy Misiak, Viorel Mititean, Jordi Mola and Pin Wu for their tremendous help in making this project a fully deployed end-to-end system.

References

[1] ApacheBench: A Complete Benchmarking and Regression Testing Suite. https://httpd.apache.org/docs/2.2/programs/ab.html.

[2] APC Index Mismatch. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x1--apc-index-mismatch.

[3] Chrome. https://www.google.com/chrome/.

[4] Event Tracing for Windows (ETW). https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[5] GDB and Reverse Debugging. https://www.gnu.org/software/gdb/news/reversible.html.

[6] JetStream 2. https://browserbench.org/JetStream/.

[7] Nginx. https://www.nginx.com/.

[8] NotMyFault. https://docs.microsoft.com/en-us/sysinternals/downloads/notmyfault.

[9] Structured Exception Handling (SEH). https://docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp.

[10] UndoDB: The Interactive Reverse Debugger for C/C++ on Linux and Android. https://undo.io/.

[11] UnixBench. https://github.com/kdlucas/byte-unixbench.

[12] Windows Debugger. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/.

[13] Windows Subsystem for Linux (WSL). https://docs.microsoft.com/en-us/windows/wsl/about.

[14] x64 Exception Handling. https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64.

[15] Gautam Altekar and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206. ACM, 2009.

[16] Prashanth P Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, pages 137–147. ACM, 2007.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual.

[19] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–32, 2018.

[20] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[21] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 2015.

[22] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[23] George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*. ACM, 2008.

[24] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[25] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX SYmposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[26] Wei Jin and Alessandro Orso. BugRedux: Reproducing Field Failures for In-House Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[27] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[28] Samuel T King, George W Dunlap, and Peter M Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, 2005.

[29] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[30] Ali Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[31] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[32] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[33] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[34] Min Xu, Rastislav Bodik, and Mark D Hill. A Flight Data Recorder for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2002.

[35] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Venkitachalam Weissman, Ganesh, and Boris Weissman. Retrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[36] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[37] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads

Gina Yuan, Shoumik Palkar, Deepak Narayanan, Matei Zaharia
Stanford University

Abstract

As specialized hardware accelerators such as GPUs become increasingly popular, developers are looking for ways to target these platforms with high-level APIs. One promising approach is *kernel libraries* such as PyTorch or cuML, which provide interfaces that mirror CPU-only counterparts such as NumPy or Scikit-Learn. Unfortunately, these libraries are hard to develop and to adopt incrementally: they only support a subset of their CPU equivalents, only work with datasets that fit in device memory, and require developers to reason about data placement and transfers manually. To address these shortcomings, we present a new approach called *offload annotations (OAs)* that enables heterogeneous GPU computing in existing workloads with few or no code modifications. An annotator annotates the types and functions in a CPU library with equivalent kernel library functions and provides an *offloading API* to specify how the inputs and outputs of the function can be partitioned into chunks that fit in device memory and transferred between devices. A runtime then maps existing CPU functions to equivalent GPU kernels and schedules execution, data transfers and paging. In data science workloads using CPU libraries such as NumPy and Pandas, OAs enable speedups of up to 1200× and a median speedup of 6.3× by transparently offloading functions to a GPU using existing kernel libraries. In many cases, OAs match the performance of handwritten heterogeneous implementations. Finally, OAs can automatically page data in these workloads to scale to datasets larger than GPU memory, which would need to be done manually with most current GPU libraries.

1 Introduction

The public cloud has commoditized specialized hardware such as GPUs, giving developers a new way to speed up their applications using these new accelerators. One increasingly popular way of doing this is to use accelerator-compatible *kernel libraries* with APIs that mirror those of popular CPU libraries. For example, in just the last two years, the Python ecosystem has seen a rapid explosion of popular GPU libraries such as PyTorch [24], cuML [8] and cuDF [7], and the RAPIDS [9] suite for data science; these libraries mirror the APIs of popular packages such as NumPy, Scikit-Learn, and Pandas. In keeping a familiar interface, accelerator libraries promise a seamless path to supporting new hardware for both new and existing applications.

Unfortunately, in reality, accelerator libraries for GPUs are not so simple to adopt into new or existing code. First, many of these libraries only implement a subset of functionality present in their CPU counterparts, e.g., because some functions are inefficient on parallel architectures such as GPUs [14, 15, 28]. This means that most applications must use *both* CPU and accelerator libraries, thus violating the promise of seamless integration with new hardware platforms. In addition, small API differences mean that even supported functions often warrant application changes. Finally, since most accelerator libraries operate over data that fits entirely in device memory, workloads in domains such as data science cannot seamlessly reap the benefits of new hardware because their working sets far outsize device memory. Developers must manually page and transfer data between the accelerator and CPU, or forego accelerators altogether.

In this paper, we propose *offload annotations (OAs)*, a new approach for incrementally integrating existing CPU libraries with emerging accelerator libraries. With this approach, an *annotator* (e.g., an application or library developer) adds annotations to CPU functions that specify a corresponding accelerator function from an accelerator library. An underlying runtime uses the annotations to automatically schedule functions either onto the CPU or the accelerator. In our system, we show that annotations enable end users to use both established CPU libraries and emerging GPU libraries without having to change their code or learn a new API. We also show that our runtime can allow end users to invoke GPU functions transparently even when data does not fit in accelerator memory. However, designing and leveraging annotations to offload computation to accelerators poses a unique set of challenges.

First, applications that mix CPU and accelerator code must be cognizant of *data placement*. For example, accelerator libraries such as PyTorch [24] can only process data resident in device memory, and GPU-resident data has an entirely different format than CPU-resident data. Our annotations explicitly keep track of the device on which a particular data value resides, and include a new API to let annotators specify how to transfer data between devices. In addition, some library functions that *allocate* new data, such as `numpy.eye()` (which creates an identity matrix), have equivalent functions in an accelerated library, so OAs let users explicitly identify such functions. The runtime uses this additional information specified in OAs to ensure that data is in the correct format on each device, helping to optimize the computation.

A second challenge in offloading functions to accelerators is *memory management*. Accelerators generally have far smaller attached memories than CPUs: this means that large CPU-resident datasets cannot naively be copied to the accelerator in entirety. To address this challenge, OAs leverage the splitting mechanism of *split annotations* [23], originally used for cross function cache pipelining on the CPU, in the new use case of *paging* memory. Our runtime partitions inputs into chunks that fit in device memory, and automatically schedules data transfer and function invocation on partitioned values. Values are partitioned and merged using a user-defined *splitting API*. By identifying splittable functions using OAs, developers can run existing CPU workloads across different platforms transparently, even if the working set does not fit in device memory. Some accelerated functions available in libraries *cannot* be split into smaller computations, however, but OAs can still offload the computation up to a certain size.

Finally, a third challenge unique to offloading functions is determining where to execute annotated functions. Since functions that execute on an accelerator must first transfer their inputs to device memory, they have an additional associated data transfer cost. This can negatively impact performance in cases where the function itself executes quickly. To address this challenge, our runtime includes a new scheduler that uses estimates of transfer cost and compute cost to determine when functions should be executed on the GPU vs. the CPU. We show that simple linear cost model estimators can yield 26× performance improvements compared to greedily executing all supported functions on a GPU.

The adoption of annotation-based approaches has already seen success in the past with systems such as TypeScript [12, 25]. In the TypeScript community, annotators crowdsource and share annotations for existing libraries. Unlike approaches that require building a complete end-to-end compiler, such as Weld [22] or Delite [31], the annotation-based approach also allows annotators to *incrementally* add support for individual accelerated functions. We hope to see a similar community develop around OAs to bridge existing CPU libraries with their accelerator library equivalents.

We implemented OAs by extending the Python runtime for split annotations, Mozart [23]. Our extended runtime, *Bach*, considers the challenges unique to offloading computation to capture device placement information and schedule computation in a heterogeneous setting. We evaluate OAs by integrating several CPU-only data processing libraries with their GPU library equivalents: PyTorch and cuPy for NumPy, cuDF for Pandas, and cuML for Scikit-learn. Our integration experience demonstrates the generality of OAs for popular data science libraries, and the minimal developer effort involved that requires little to no code modifications. On data science workloads ranging from options pricing to principal components analysis, OAs are able to achieve up to 1200× improvement with a median 6.3× over CPU-only code, with few or no application changes. OAs also enable many workloads to use GPUs when the dataset size exceeds the GPU memory, which would require manually paging code with the existing GPU computation libraries.

```
# Fit.
m1 = sklearn.StandardScaler()
m2 = sklearn.PCA() # or cuml.
m3 = sklearn.KNeighborsClassifier() # or cuml.
X_train = m1.fit_transform(X_train)
+ X_train = transfer(X_train, GPU)
X_train = m2.fit_transform(X_train)
+ Y_train = transfer(y_train, GPU)
m3.fit(X_train, Y_train)
+ for chunk in f:
  # Predict.
  X_test = m1.transform(X_test)
+ X_test = transfer(X_test, GPU)
  X_test = m2.transform(X_test)
  result = m3.predict(X_test)
+ result = transfer(result, CPU)
plottinglib.plot(result)
```

Listing 1: Example data science workload using `sklearn`. Lines preceeded with a **+** show modifications required for using a GPU with the `cuML` accelerator library.

In summary, we make the following contributions:

- We introduce offload annotations (OAs), an interface for heterogeneous computing with no library modifications that allows third-party annotators to incrementally add accelerated versions of library functions, and manages the offload and execution of these functions on devices. The OA interface extends split annotations with support for representing data in different formats on different devices and deciding when to offload a computation based on its estimated transfer size and computation cost.

- We describe Bach, a Python runtime that uses annotations to capture a lazy task graph of program operations and schedules execution and data transfer, including allocations, in order to improve application performance while staying within the accelerator's memory limits.

- We integrate OAs with four kernel libraries for GPU computation, and show that they can accelerate applications by a median of 6.3× over the CPU-only version of the library. We also compare the performance of OAs to hand-written heterogeneous code that manually combines these libraries with CPU libraries.

2 Motivating Example

To motivate the OA approach, consider the following simple scenario: A data scientist has a machine learning workload originally written for the CPU using `sklearn`. She reads the data from a file on disk, preprocesses the data, trains the model, and then tests it on a real dataset (Listing 1).

As her company sends her more and more data, the data science pipeline takes an order of magnitude longer to run. She learns about accelerators and accelerator libraries such as cuDF and cuML built to speed up data science workloads using GPUs. Since the pipeline was already running on cloud instances, the data scientist decides to move her code to another instance with accelerators attached.

The online documentation for these kernel libraries promises a seamless integration experience, offering almost exactly the same interface as their CPU library counterparts, but she soon discovers it is not as easy as it seems. Some of the functions have different names, requiring her to scour the documentation for functions with the corresponding functionality. Some functions do not have corresponding implementations at all, and can run only with the CPU library.

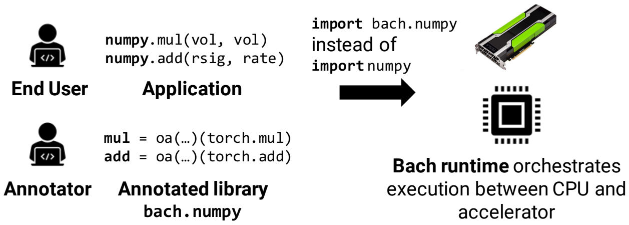
Next, the data scientist analyzes the program and manually inserts data transfer statements between the GPU and the host CPU so that data resides on the same device as the corresponding functions. Since the inputs to the program are usually read from another step in the pipeline, she assumes the inputs initially reside on the CPU. Since she initializes some of the intermediate objects herself, she allocates those directly on the GPU. Finally, since her plotting library requires the data to be on the CPU, she transfers the results back to the CPU at the end of the program.

With most CPU library functions replaced with the corresponding GPU library functions, she is ready to run the program, but it crashes due to insufficient GPU memory. Although her dataset was small enough to fit in CPU memory, the amount of memory attached to the GPU is significantly smaller. She writes additional code to page the data transfers to the GPU in the prediction phase, since the `predict` computation can be done independently on different data batches. The training function in the accelerator library *cannot* be run in independent batches, but fortunately, her training data is smaller than the data she predicts on, so that computation can run as one GPU function call.

Finally, after all the developer effort required to learn and integrate the GPU libraries, the code runs to completion, and the data scientist receives the same results on her dataset as when the pipeline ran only on the CPU. The performance of her new program has also improved. Nonetheless, the data scientist's code is now complex (Listing 1), with many new function calls and new control logic just to manage the GPU computation. Furthermore, all this new logic may need to change in the future as her workload changes or her dataset changes in size.

3 Design Overview

With *offload annotations* (or *OAs*, Figure 1), we reduce the developer effort for porting an existing workload to the GPU to just importing the annotated CPU library in place of the original CPU library. The annotator could be the kernel library developer, the end user writing applications, or any other third-party developer (similar to the open source contributors that provide type definition files for libraries in TypeScript [12]). Our system, Bach, uses the information in OAs to automatically offload functions to a GPU, page large datasets, transfer data across devices automatically, and manage allocations to minimize data transfer for better performance.



```
numpy.mul(vol, vol)
numpy.add(rsig, rate)
```

```
mul = oa(…)(torch.mul)
add = oa(…)(torch.add)
```

```
import bach.numpy
```
instead of
```
import numpy
```

End User
Application
Annotator
Annotated library
bach.numpy
Bach runtime orchestrates execution between CPU and accelerator

Figure 1: Overview of writing and using OAs. An *annotator* writes annotations to bridge a CPU library with an accelerator library. An *end user* imports the new annotated library to automatically use new accelerators in her existing code.

Adding OAs to CPU libraries. First, an annotator identifies a corresponding accelerator library for her CPU library (e.g., `torch` for `numpy`). She then identifies a corresponding accelerator function for each CPU function she wishes to annotate (e.g,. `torch.multiply` for `numpy.mul`). The annotator then writes an OA for her CPU function: the OA specifies the corresponding GPU function that should be called in place of the CPU function, and split types for each function input and output (adapted from the split types used in split annotations [23]). Split types are an interface implemented by an annotator that provide information about a function input or output *at runtime* (e.g., the size of an array). The annotator can also write special OAs for allocation functions (e.g., `torch.zeros` for `numpy.zeros`), which enable data to be allocated directly on the device where they will first be used.

OAs extend the original split type interface to provide additional information about *data placement*. In particular, the annotator must implement a new *offloading API* for each split type. Most libraries only require implementing the offloading API once per *data type* in the library (e.g., for `ndarray` in NumPy). The offloading API specifies (1) where inputs to a function reside before execution (e.g., in GPU memory or CPU memory), and (2) how to *transfer* values from from one device to another. Since the offloading API extends the split annotation splitting API, it can also *optionally* describe how to split and merge data for data-parallel workloads. Splitting in OAs is used for paging data into an accelerator with limited memory (unlike in split annotations, where splitting enables cross-function cache pipelining on a CPU).

Once an annotator adds OAs and implements the offloading API for the data types in the CPU library, she can share the annotation file to allow other end users to reap their benefits.

Using annotated libraries. An *end user* integrates the annotated library into her code by changing the line that imports the CPU library to import the annotated library instead (the annotation file is just a Python module).

Generally, little to no code modification is required to use the annotated library in place of the original CPU library. The main difference is that Bach, our runtime, uses *lazy evaluation* to optimize data transfer across many functions. OAs can automatically evaluate lazy values in many cases (e.g., when calling `str()` in Python), but an end user may have to add `evaluate()`—a function automatically provided in annotated libraries—into her code to explicitly materialize lazy values.

Bach runtime scheduler. Bach builds on split annotations' Mozart runtime to automatically build and maintain a lazy task graph. Internally, when a lazy value is materialized, Bach uses OAs to automatically schedule data transfers and computation, deciding the device on which to run each operation (§5). Note that scheduling is completely packaged in our runtime, and does not require any additional annotator or end user code.

Although the Bach runtime defaults to greedily scheduling operations on the GPU, the annotator may still provide custom cost model estimators to the function annotations to assist the runtime with making scheduling decisions. However, these cost models are optional and are not usually required to benefit from OAs, as shown in our evaluation.

4 Offload Annotation Interface

The offload annotation (OA) interface provides a corresponding accelerator function for each function in a CPU library. The interface also provides a mechanism to discover runtime information about function arguments and outputs: namely, how to split inputs into chunks that will fit in device memory, the device on which an input is allocated before executing annotated functions, and how inputs can be transferred between devices. This information is relayed via *split types*, an abstraction from split annotations [23]. The OA interface also includes new *`alloc` annotations*, which specify functions that allocate new data (e.g., `numpy.zeros` to allocate an array of zeros). These annotations allow further optimizations when scheduling data transfer and are described further below.

Listing 2 shows an example of the extended offload split type API, and Listing 3 show examples of OAs, with `numpy` as the CPU library and `torch` as the accelerator library.

4.1 Primer: Split Types

Split types provide a mechanism that allow a runtime to discover runtime properties about a value (e.g., size of an array or dimensions of a matrix). In split annotations, split types are used to ensure that data is *split* in a consistent way across functions to enable safe pipelining of split values. For example, a split type will ensure that split arrays passed into a function together still have the same lengths at runtime.

```
class DataFrameSplit(OffloadSpliType):
  def split(start, end, value):
    # Splits a value to enable paging.
    return value[start:end]
  def merge(values):
    # Merges split values
    return pandas.concat(values)
  def size(value):
    # Returns number of elements in value
    return len(value)
  def device(value):
    # Specifies where this value is allocated.
    # Used by scheduler to decide where to run
    # functions.
    if isinstance(value, pandas.DataFrame):
      return Device.CPU
    else: # if a cuDF DataFrame.
      return Device.GPU
  def to(value, device):
    # Transfers [value] to specified [device].
    if device == Device.GPU:
      return cudf.from_pandas(value)
    else:
      return value.to_pandas()
```

Listing 2: Example implementation of the offload split type API for Pandas and cuDF DataFrames. The API adds two new functions—`device` and `to`—to the original split type API.

To use split types, an annotator implements an API that the runtime calls to interact with runtime values. In split annotations, split types provide a `split` function to partition values into chunks, and a `merge` value to merge split values together. The API also contains a `size` function for discovering the size of inputs (e.g., to determine split sizes). Listing 2 shows an example of these functions for DataFrames. We extend the split type API to allow our runtime to offload values onto other devices.

When splitting and merging data in a non-trivial way, end users must ensure the correctness of the application. Many data science applications operating on large collections of data, as in our workloads, are trivially parallelizable. Even when splitting and merging is impossible due to algorithmic correctness constraints or unavailable kernel library implementations, applications can still benefit from automatic offloading at smaller data sizes.

4.2 Offload Split Types

In addition to specifying how data should be split and merged, our extended *offload split types* additionally specify *(1)* the device on which a value is allocated, and *(2)* how to transfer a value between devices.

`Device` API. The `device` method specifies the device its input resides on. The method might check the instance type

```
# Offload split types for binary function inputs.
args = (NdArraySplit(), NdArraySplit())
# Offload split type of return value.
ret = NdArraySplit()

# OAs to provide offload split types for each
# argument and return value, as well as a corresonding
# accelerator function.
np.add = @oa(args, ret, func=torch.add)
np.subtract = @oa(args, ret, func=torch.sub)

# Allocation function.
np.empty = @oa_alloc(NdArraySplit(), func=torch.empty)
```

Listing 3: Offload annotations using `numpy` and `torch`.

of the input, or properties of the input. For example, NumPy arrays are on the CPU while CuPy arrays are on the GPU. Torch tensors can reside on either device, and have a property to describe where a particular value resides.

`To` API. The `to` method transfers the provided value to the target device. This usually involves converting a CPU library type (e.g., `numpy` array) to an accelerator library type (e.g., `torch` tensor) using a accelerator library function (e.g., `torch.to()`). The ability to transfer values is necessary to ensure that values reside on the device where the operation will run.

4.3 Using Offload Split Types in Annotations

OAs assign each input and output an offload split type. Additionally, the OA provides the name of the corresponding accelerator library function. If the accelerator function has a different function signature, an annotator can wrap the accelerator function in a lambda with an interface consistent with the CPU function.

Example. Listing 3 shows several examples where NumPy functions are annotated using PyTorch. The OAs assign the two arguments of `np.add` and `np.subtract` the offload split type `NdArraySplit`. This split type will define how to split, merge, and transfer `ndarray` and `torch.tensor` values. It will also tell Bach whether a particular value passed to these functions is already on the accelerator or CPU using the `device` API. The outputs of these functions also have the offload split type `NdArraySplit`.

4.4 Allocation Function Annotations

One unique challenge the runtime faces is avoiding unnecessary data transfers, which can lead to performance degradation. Consider when data is allocated on one device, and then immediately passed to a function that can be offloaded. In this case, it is more efficient to allocate the data directly on the device of the following function.

To support this, OAs provide a special kind of annotation, called an `alloc` annotation, which specify that the annotated function performs allocation. Like other annotated functions, annotators provide CPU allocation functions with an equivalent accelerator library functions (e.g., `torch.zeros` for `numpy.zeros`). Allocation functions differ from regular functions in one key aspect: their inputs do not need to be annotated with offload split types. Outputs are still annotated with a offload split type, similar to a regular function. CPU-only split annotations did not require allocation annotations since they did not need to avoid expensive data transfers.

Example. In Listing 3, the `np.empty` function allocates data: its OA specifies a corresponding PyTorch function, but does not provide offload split types for inputs. Its output has the same `NdArraySplit` offload split type.

5 Bach Runtime

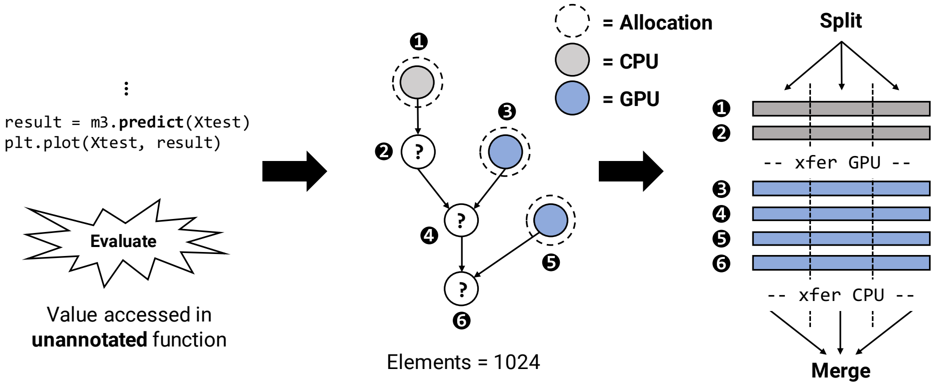
The Bach runtime uses the information in the OAs to schedule and execute functions across the CPU and accelerator. Figure 2 provides an overview of the Bach runtime.

Step 1: Construct Dataflow Graph. Bach's first goal is to extract a dataflow graph from the user program. To do this, Bach uses the same approach as Mozart: when annotators apply an annotation to a function, Bach wraps it to return a lazily evaluated placeholder object, using Python's metaprogramming facilities [23]. When placeholders objects are passed to other annotated functions, Bach stitches these functions into a task graph. This task graph captures dependencies between annotated functions: an edge between two operations exists if the output of one operation is used as an input into another.

Two scenarios trigger evaluation of the task graph. First, Bach detects accesses to the lazy placeholder objects by intercepting certain Python methods (e.g., `str` to convert an object into a string, or `__getitem__` to index into a collection). Internally, the placeholder object will trigger evaluation of the full task graph required to build it, and then forward these method invocations to the evaluated object. For example, this means that placeholder objects will be evaluated if the user passes them to `print()`. Alternatively, the user can also explicitly call an `evaluate()` function to trigger execution.

Step 2: Estimate Data Size and Allocate. In order to correctly partition the data for splitting, Bach must estimate the data size by using the `size()` API on the program inputs. Unlike in Mozart, Bach can lazily allocate values to optimally place data on the same device as the first function that uses that value. However, if all the program inputs are lazy allocations, there are no available values with which to estimate data size. Thus Bach must allocate lazy values before starting execution to estimate the data size.

Bach decides where to allocate each lazy value based on the device of the first function to use the value. OAs differ from split annotations in this regard because they need to decide which device to run each function on. A function can



```
⋮
result = m3.predict(Xtest)
plt.plot(Xtest, result)
```

Figure 2: Overview of the steps involved in Bach's runtime. Step 1 triggers evaluation of the dataflow graph with an access to a lazy value. Step 2 allocates the lazily allocated nodes and estimates the program data size. Step 3 dynamically schedules the instructions across devices, inserting data transfers and paging the inputs.

```
# Heuristic for estimating data transfer cost.
def transfer_estimator(ty, values, device):
    x = ty.size(values)
    return a * x + b

# Heuristic for estimating compute cost.
def compute_estimator(ty, values, device):
    x = ty.size(values[0])
    if device == CPU:
        return a_cpu * x + b_cpu
    else:
        return a_gpu * x + b_gpu
```

Listing 4: Linear estimators for estimating data transfer and compute cost.

run on the accelerator if it has an `oa` or `oa_gpu` annotation, and if its inputs either can be transferred to the device or already reside on the accelerator. All functions must be able to run on the CPU, which is the default device.

To decide where to run this first function, Bach estimates the data transfer costs and compute costs involved with running the function on either device and suggests the device with the lower cost. These cost estimates are calculated using heuristic functions optionally provided by the user. The heuristic functions are functions of the input values, their offload split types, and the target device, and we provide a simple linear cost model estimator (Listing 4). If cost models are not provided or all other inputs are also lazy allocations, Bach naively suggests the function run on the accelerator if possible. Otherwise Bach defaults to the CPU.

Step 3: Schedule and Execute. Once the data size is estimated, the operations in the task graph are converted into a list of instructions that can be executed serially for each split piece. To do this, the Bach runtime performs a topological sort of the task graph to obtain a list of instructions that satisfy data dependencies.

The device an instruction runs on is decided dynamically right before executing the instruction. The instruction first determines if it is eligible to run on the accelerator based on the requirements described in the previous section. If it is not eligible, it defaults to running on the CPU. Otherwise, the runtime performs the same cost model analysis as when deciding where to lazily allocate a value to determine which device to run the instruction on. Unlike before, all the inputs will be fully evaluated. After deciding which device to use, the instruction transfers inputs that are on a different device using the `to()` API in the offload split type for the input. Bach discovers where inputs are prior to executing functions by using the `device()` API. As before, if cost models are not provided, Bach defaults to using the accelerator if possible.

When executing functions, Bach uses the ability to partition data to enable *paging*: this allows for large, CPU-memory-resident datasets to be streamed through device memory, even when device memory is far smaller than the CPU memory. To achieve this, Bach splits the inputs into chunks based on the estimated data size and a default piece size. Inputs are split using the `split()` API provided in the input's offload split type. For each chunk, the program executes the generated list of instructions. The chunk is transferred to the device of the input argument if its current device does not match the device of the instruction. Each chunk is moved out of the device after the functions finish executing, to free space for the next chunk.

The final outputs are moved to the CPU by default after execution, but the end user can elect to keep the output on the accelerator by explicitly calling `evaluate()`. If paging is used to stream data through a device, the output is always allocated on the CPU (since it may not fit entirely in device memory).

6 Design Discussion

As described, OAs and the Bach runtime are designed specifically for offloading computation to a single GPU using Python kernel libraries. In this section, we discuss the possibilities of extending OAs for use with multiple GPUs or with other programming languages and accelerators.

Multiple GPUs. Similar to how split annotations split data-parallel workloads across CPU cores, we can extend OAs to automatically split computation across multiple GPUs. The implementation would need to modify the scheduler to recognize multiple GPU targets, and factor data transfer and concurrency into scheduling decisions. We do not expect these modifications to impact end user experience.

Non-Python programming languages. We chose to implement Bach in Python since Python is one of the most popular languages for data analysis, with a vast ecosystem of Python GPU libraries. We believe our implementation uses principles common to many programming languages and do not rely on any language-specific hacks. Specifically, "annotations" in Python are simply functions that wrap other functions, and the runtime logic is language-agnostic. Mozart [23] demonstrates that the annotation framework is viable in C++, so we imagine we could implement a similar runtime for C++ for GPU libraries like Thrust.

Non-GPU accelerators. Any accelerator with a kernel library that closely mirrors a CPU-only library and an API to offload data to that accelerator could potentially be suitable for OAs. We may also be able to adapt split data in streaming accelerators to overlap data transfer with computation like in CUDA streams. Data transfer costs are an issue common to many accelerators, and we could apply ideas about memory management and data placement from OAs even if the system cannot be used directly.

7 Library Integrations

We annotated three different CPU-only Python libraries for data science and machine learning: NumPy, Pandas, and Scikit-learn. The annotations used four different GPU kernel libraries: CuPy, PyTorch, cuDF, and cuML. The latter two kernel libraries are part of the RAPIDS [9] suite.

NumPy. NumPy is a library for high-level math operations on multi-dimensional arrays and matrices on the CPU. NumPy was the most popular library in terms of number of accelerator library equivalents. In their online documentation, these accelerator libraries directly claim to provide a NumPy-like API. CuPy is described as a NumPy-like API accelerated with CUDA, while PyTorch is described as a replacement for NumPy that leverages the power of GPUs. We integrate NumPy with CuPy and PyTorch in two separate OA-libraries, replacing NumPy `ndarrays` with CuPy `ndarrays` and PyTorch `tensors`.

CPU-only library	GPU kernel library	LOC	# Split Types	# Funcs
NumPy	CuPy	103	1	20
NumPy	PyTorch	90	1	10
Pandas	cuDF	241	7	27
Scikit-learn	cuML	81	2	12

Table 1: Integration effort for annotating various libraries. Lines of code include annotations, split type transfer functions, and splitting functions.

Pandas. Pandas is a data analytics library for operating on structured table-like or time series data. The cuDF accelerator library provides a Pandas-like API. We replace Pandas `DataFrame` and `Series` data types with the corresponding cuDF types.

Scikit-learn. Scikit-learn is a popular machine learning library. Machine learning is a natural fit for the GPU with its dense linear algebra operators. The cuML library's Python API attempts to closely match the Scikit-learn API.

7.1 Integration Experience

From our experience, library integrations required around 130 lines of code per library (Figure 1). The most lines of code come from implementing the offload split type API for transferring, splitting, and combining types. However, the split and combine API is optional if a user does not need to page large datasets. In the simplest and most common case, an OA requires only a single line of code per function to specify the offload split types of inputs and outputs, and the name of the kernel library function. These annotations resemble boilerplate code when libraries repeat a common pattern, like binary operations with a single output in NumPy. In more complex function annotations, the benefit of OAs is that it only needs to be done once in the annotated library as opposed to every instance in every workload.

7.1.1 Straightforward Drop-In Replacements

Every library has a straightforward way to transfer data to the appropriate device. Most accelerator library types automatically reside on the GPU, so using CuPy's `ndarray` or cuDF's `DataFrame` automatically transfers the data to the GPU. Scikit-learn can use either CuPy's `ndarray` or cuDF's `DataFrame` as the underlying representation. PyTorch tensors can reside on both CPUs and GPUs, so the transfer implementation from Numpy `ndarray` first converts the `ndarray` to a `torch.Tensor` (a zero-copy cast) and then calls a method on the tensor to transfer it to the GPU.

Just as many accelerator libraries claim to closely resemble the CPU libraries they attempt to replace, many accelerator library functions are indeed a drop-in replacement for the corresponding CPU library function. For example, numpy.add(), cupy.add(), and torch.add() are the same across all three libraries. Sometimes, the method names are trivially different but represent the same functionality, like numpy.arcsin() and torch.asin(). Scikit-learn's API utilizes a complex module structure that does not exist in cuML (e.g., sklearn.decomposition.PCA vs cuml.PCA), so annotators sometimes must mock module structure to make integration as seamless as possible.

7.1.2 Different Function Specifications

Even if two CPU and kernel library functions appear to be equivalent based on name, the annotator must be careful to ensure the function parameters and specifications are identical. For example, the array allocation functions numpy.ones() and torch.ones() both take a parameter dtype to specify the data type of the array. However, NumPy can accept strings like 'int8' as a parameter, while PyTorch only accepts library-defined types like torch.int8. In this case, the annotator must write a custom wrapper that converts function parameters.

We experienced another challenge involving different function specifications when integrating Pandas with cuDF. Both pandas.read_csv() and cudf.read_csv() read a CSV into a DataFrame object. In Pandas, the squeeze parameter causes the function to return a Series if the parsed data only contained one column. To achieve the same functionality in cuDF, which does not have this parameter, we wrote a custom function that converted the returned DataFrame into a Series if the squeeze parameter was included. Of the 48 parameters in v0.25.2 of pandas.read_csv()'s documentation, others are also bound to not exactly match the specifications in cuDF and require special implementation.

7.1.3 Missing Functions

When a function is missing from an accelerator library, any library annotator can annotate the library with a custom function. We implemented a custom GPU version of the Pandas mask() function, used for replacing values in a DataFrame based on a conditional DataFrame, by using the series.loc[cond] = val notation from the cuDF library instead. In our cuML annotations, we mimicked Scikit-learn's StandardScaler() model using CuPy operations for removing the mean and scaling to unit variance on the GPU. In general, cuML's preprocessing library is far behind Scikit-learn's, potentially because Scikit-learn's functionality is easy enough to achieve with other accelerator library functions.

In other cases, functions do not exist because the algorithm required to provide the functionality is either impossible or simply unsuitable for the GPU. In particular, many Pandas functions do not work in cuDF when string operations are involved, requiring the program to execute on the CPU. Though libraries like nvStrings and cuStrings are working to close the gap in text processing on the GPU, the state of strings on the GPU today still requires the developer to have a deeper understanding of its literal representation on the GPU.

Workload	Ops	CPU Library	Max Speedup
Black-Scholes	39	NumPy[1]	5.7×
Black-Scholes	39	NumPy[2]	6.9×
Haversine	19	NumPy[1]	0.81×
Haversine	19	NumPy[2]	1.7×
Crime Index	15	Pandas	4.6×
DBSCAN	7	NumPy[1]/Sklearn	1200×
PCA	8	Sklearn	6.8×
TSVD	2	Sklearn	11×

Table 2: The evaluated workloads, the number of annotated function operators, and the CPU Python libraries used by each workload. The median speedup across workloads is 6.3× with a maximum speedup of 1200× on DBSCAN. Annotated with CuPy[1]. Annotated with PyTorch[2].

7.1.4 Multi-Library Integration

Just as libraries in the Python data science ecosystem use each other in their implementations, annotated libraries must also be able to import and operate on other annotated libraries. In the CPU ecosystem, Scikit-learn's functions use the NumPy ndarray, while in the GPU ecosystem, cuML's functions use the CuPy ndarray. In our annotated Scikit-learn library, we analogously import the NdArraySplit type from our annotated NumPy library to define the argument and return types in the OAs. As OAs grow in popularity, we imagine an ecosystem of increasingly-interconnected annotated libraries that allow seamless execution across multiple devices in existing workloads.

8 Evaluation

We ran experiments on a 56-CPU server (2× Intel E5-2690 v4) with 512GB of memory, running Linux 4.4.0. The machine has a single NVIDIA Tesla P100 GPU with 16GB of memory and CUDA 10.2 installed. Each result is the median of five runs, with one warm-up run omitted to initialize the CUDA driver. Workload runtimes are measured end-to-end, including allocation and synchronization operations at the end. Our source code is available at https://github.com/stanford-futuredata/offload-annotations.

8.1 Workloads

We evaluated offload annotations on a variety of workloads adapted from common mathematical formulas, data science library tutorials, and popular online blog posts (Table 2).

Black-Scholes [1]. Determines the theoretical value for a large array of call or put options.

Haversine [3]. Determines the great-circle distance between points on a sphere.

Crime index [4]. Reads population and robbery data from a file on disk and calculates a numerical crime index score.

DBSCAN [5]. Standardizes a dataset with 256 features around 32 centers, and clusters the data with the DBSCAN algorithm, analyzing the predicted labels on the CPU.

PCA [6]. Standardizes training data and reduces the dimensionality with PCA, then classifies the points with K-Neighbors. Predicts the results, followed by plotting.

TSVD [10]. Applies the linear dimensionality reduction algorithm to a synthetic dataset with 512 features.

8.2 Results

Figure 3 showcases the performance of the data science and machine learning workloads on inputs of various sizes. We evaluate the workloads on a CPU-only implementation, a handwritten accelerator kernel implementation, and an annotated implementation with Bach.

We verified the numerical results of the workloads for correctness against each implementation. In the machine learning workloads, we selected parameters that produced reasonable results given the inputs (e.g., DBSCAN predicted 32 clusters and PCA classified points with greater than 90% accuracy). We maintained the same parameters for each implementation to ensure the parameters did not affect the runtime.

The results show that with less developer effort, Bach is able to match the performance of handwritten GPU implementations, scale to larger input data sizes that normally cause the GPU to run out of memory, and outperform CPU library implementations. We now discuss these results in more detail.

8.2.1 Results Summary

We make three main observations.

First, Bach matches the performance of handwritten GPU implementations in all workloads. Bach selects which CPU library functions to offload and how to transfer data, with minimal developer effort.

Second, Bach scales to larger input data sizes that normally cause the handwritten GPU implementations to run out of memory. In most workloads, Bach's performance continues to scale at the same rate as before the GPU implementation runs out of memory, giving the appearance of running the same program on a GPU with infinite memory. DBSCAN is the only workload that cannot be split since the clustering model must be fit on all data at once. However, this is not a significant limitation since DBSCAN also has the largest runtime, with Bach taking 7.5 hours to run at the largest piece size before running out of memory, and the CPU implementation taking considerably more for the same input size.

Third, Bach outperforms CPU implementations for almost all workloads. One exception is Haversine Torch, which has worse performance than the CPU implementation when paging large datasets due to the overhead from the additional data transfers. Another exception is PCA at data sizes below 2^{12}. We attribute this to overheads associated with initialization and launching GPU kernels with significant launch overhead. CPU implementations outperform Bach for other workloads as well, but only for small input sizes when runtimes are in milliseconds or less.

8.2.2 Runtime Distributions

Workloads that benefit from GPU acceleration are suitable for acceleration using OAs. At a high level, the decision to use the GPU is a tradeoff between expensive data transfers and faster compute. The impact of OAs, and more broadly the impact of the GPU, depends on the distribution of runtime between data transfer, computation, and memory allocation in each workload (Fig. 4).

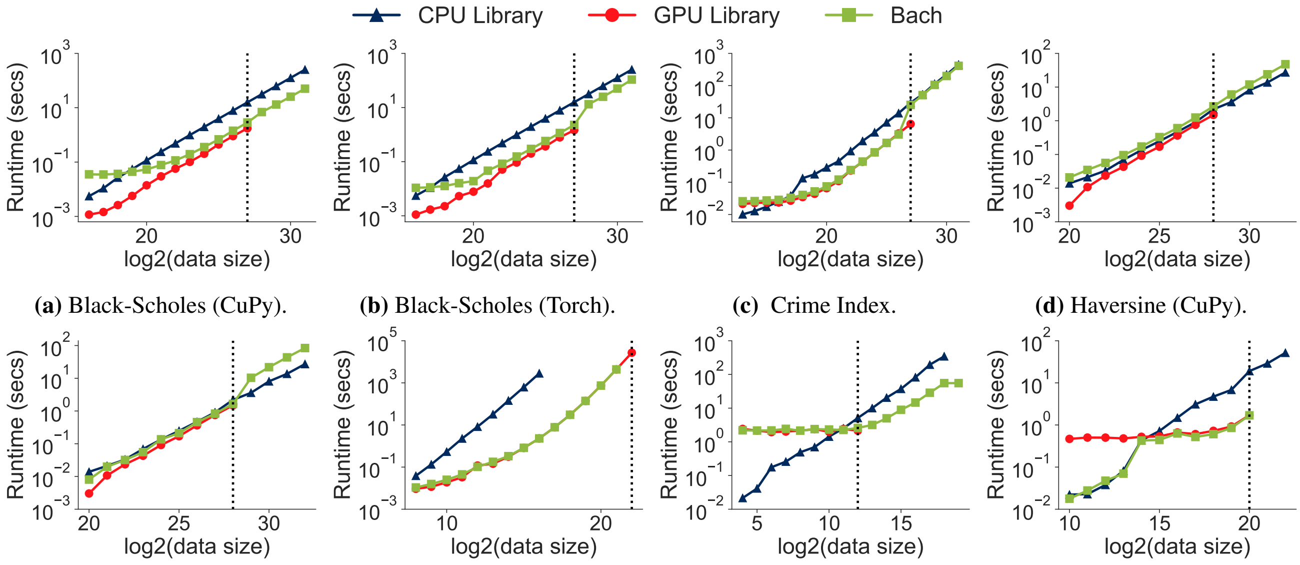
Allocation. Crime Index spends 93% of its total runtime of 29.74s on allocating data. In particular, the workload reads 1GB of data into memory before performing a series of fast numerical operations and calculating a single number, the crime index, as an output. Managing memory allocations are particularly important for expensive I/O operations like reading files from disk. Workloads with these kinds of operations particularly benefit from lazy allocation.

Though allocation can be fast on either device, the primary performance benefit of lazy allocation is eliminating the additional data transfer required to move the input to the appropriate device. At large data sizes that do not fit in GPU memory, the initial memory must instead be allocated on the CPU where it does fit and paged into GPU memory using data transfers. As shown by the dotted lines in Fig. 3, beyond a specific data size, Crime Index, Haversine Torch, and Black-Scholes Torch all have additional overhead when paging large data sets due to the extra initial data transfer.

When the Crime Index (Fig. 3c) dataset fits in GPU memory and the workload is able to use lazy allocation, an additional million rows of data increases the runtime by only 50 ms, compared to 190 ms when the dataset does not fit in GPU memory. The dataset size exceeds GPU memory beyond $2^{27} \approx 100$ million rows. We calculated these overheads as an average of the scaled Bach implementation runtimes for log data sizes 21-26 and 27-31, respectively. We do not consider data sizes below 2^{21} to discount the small absolute scheduler overheads that are independent of data size.

Data Transfers. Haversine CuPy and Black-Scholes CuPy spend 52% and 60% of their total runtimes of 2.85s and 2.60s on data transfers (Fig. 4). Both workloads apply a sequence of fast numerical operations on large arrays initialized on the CPU, which must be transferred to the GPU. The workloads output more large arrays which must be transferred back. However, Black-Scholes still beats the CPU library implementation with Bach at all sizes, meaning larger absolute performance gains at larger input sizes.

As the less computationally-intensive workload, Haversine is more significantly affected by additional data transfers when paging large datasets, no longer beating the CPU implementation at large dataset sizes. It should be noted that in



(a) Black-Scholes (CuPy).

(b) Black-Scholes (Torch).

(c) Crime Index.

(d) Haversine (CuPy).

(e) Haversine (Torch).

(f) DBSCAN.

(g) PCA.

(h) TSVD.

Figure 3: Runtime vs. input data size for the workloads in Table 2. Bach is able to match the performance of the corresponding GPU library for most applications. The framework in parentheses refers to the GPU library used by Bach in the workload. The dotted line represents the input data size at which the GPU runs into an out-of-memory exception; Bach is able to run workloads past this size by paging chunks in and out of GPU memory. (Log2 piece size = 27, 27, 21, 21, 26, 22, 12, 20 from (a-h))

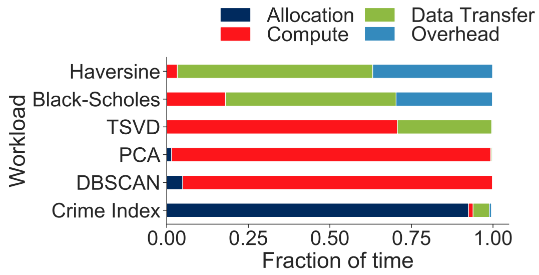


Figure 4: Proportion of total runtime split between memory allocation, data transfer, compute time, and runtime overhead in various workloads using Bach. Haversine and Black-Scholes mostly consist of data transfers; PCA, DBSCAN, and TSVD are compute-heavy; Crime Index spends the most time on allocation. (Log2 data size = 27, 28, 27, 19, 12, from top to bottom.)

absolute terms, the total time spent on data transfers for these workloads is relatively small. However, it is still important to optimize wherever possible, since the overheads can be exacerbated at scale or the applications using these pipelines might require real-time results.

Computation. DBSCAN and PCA, on the other end of the spectrum, are computationally-intensive workloads that are highly optimized for the GPU. In DBSCAN, data transfer took less than 1% of the total runtime of 130.81s, compared to the 94% spent on compute (Fig. 4). PCA spent 95% of the total runtime of 1.02s on compute. Machine learning models, with their parallelizable and computationally-intensive numerical operations, are particularly suited for the GPU.

8.2.3 Scheduling

The dynamic scheduling algorithm has minimal effect when all program inputs can be lazily allocated. In this case, all program inputs start out on the GPU if possible, even if the data size is small. Even though execution would have been been faster exclusively on the CPU, it is no longer worth transferring the inputs back to the CPU due to the overheads involved. This occurs in all workloads except the machine learning workloads: DBSCAN, PCA, and TSVD.

Among the machine learning workloads, we provided cost estimators to the TSVD workload to enable dynamic scheduling. DBSCAN already optimally executes the entire program on the GPU for all data sizes, and would not benefit from dynamic scheduling. We did not evaluate the dynamic scheduler on PCA, though it exhibits a similar runtime profile to TSVD.

In the TSVD workload, we used the linear estimators in Listing 4 as heuristics for transferring the input `ndarray` and computing the model's `fit()` function. In the transfer estimator, we use parameters `a = 1` and `b = 0` to indicate that data transfer is proportional to the size of the data regardless of device. In the compute estimator, we selected parameters based on the equilibrium point between and the CPU and kernel library lines in Figure 3h. We use `a_cpu = 2` and `b_cpu = 0` to indicate that on the CPU, computation is highly correlated to input size. For the GPU, we use `a_gpu = 0` and `b_gpu=14` to indicate that the computation is extremely cheap but incurs kernel launch overheads.

The result of using these linear estimators is a threshold scheduling algorithm that causes the total runtime of TSVD to be the minimum of the CPU library implementation and GPU library implementation (Fig. 3h). The scheduler switches from greedy GPU scheduling to CPU-only scheduling below the equilibrium data size 2^{14}. Below this data size, the GPU implementation remains constant at around 500 ms, while the CPU and Bach implementations become as low as 20 ms for data size 2^{10}. Thus Bach improves the TSVD runtime by up to 480 ms with the dynamic scheduler estimators, as it otherwise would have defaulted to using the kernel library.

It should be noted that in these workloads, the dynamic scheduler only generates an advantage at smaller data sizes where the overhead of GPU initialization is more pronounced. CPU libraries also tend to benefit from cache performance cliffs, which is why the CPU runtimes are not perfectly linear at smaller data sizes. In these cases however, the absolute runtime improvements from using the dynamic scheduler are small. In general, workloads do not require the annotator to implement any cost model at all to benefit from OAs especially at larger data sizes, where the execution is more likely to perform better on the GPU with the greedy scheduler.

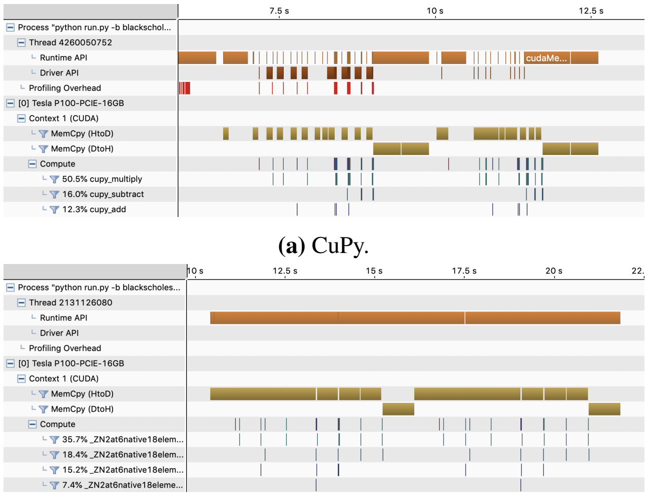
8.2.4 Discussion

Our experience integrating multiple kernel libraries with their CPU library equivalents gave us several interesting insights into the existing Python ecosystem for GPUs, including the lack of GPU kernels for several CPU functions and the differences in seemingly identical kernel library implementations.

Missing GPU implementations. Black-Scholes CuPy and PCA both contained functions without kernel library equivalents, the `numpy.erf()` error function and `sklearn.StandardScaler()`, respectively, despite these functions being trivial to run on the GPU. We addressed these performance issues in different ways.

In Black-Scholes, we wrote a custom GPU function to annotate `numpy.erf()` and eliminate additional data transfers that disrupted a numerical analysis pipeline that otherwise ran completely on a GPU. We copied the implementation from a file in CuPy's experimental folder for SciPy routines. This example demonstrates that regardless of whether the kernel library developer or third-party annotator contributes to an annotated library, annotations can make it easier to incrementally add support for GPUs into an existing workload.

Though we could have implemented a custom GPU function for `sklearn.StandardScaler()`, we did not do so because



(a) CuPy.

(b) PyTorch.

Figure 5: The NVIDIA Visual Profiler visualization of Black-Scholes annotated with CuPy and PyTorch, when paging large datasets into GPU memory. (Data size = 2^{28}; Piece size = 2^{27})

the compute time for this numerical preprocessing step was small in comparison to the prediction part of the workload, and the performance impact would not have been significant.

CuPy vs PyTorch. We were able to observe subtle differences between CuPy and PyTorch when integrating them with the NumPy data science library. When paging large datasets, annotated PyTorch incurred a higher overhead from the additional data transfers compared to annotated CuPy. Using the NVIDIA Visual Profiler, we observed that while PyTorch explicitly executed the memory transfers and kernels that we invoked, CuPy was significantly more complicated (Figure 5). In particular, CuPy made several calls to functions like `cudaHostAlloc()` and `cuModuleLoadData()`. These extra functions may have allowed CuPy to perform more efficient data transfers when paging large datasets, incurring less overhead.

9 Limitations

Without a more sophisticated scheduling algorithm, the end user may not get optimal performance using an OA-annotated library since all possible annotated functions will execute on the accelerator by default. Some functions, such as specific machine learning algorithms or analytics operations like joins, may be more efficient on the CPU in some cases. Although GPU scheduling is a complex problem, we found that greedy scheduling was effective in many applications. Though we cannot definitively state whether a holistic scheduler is better, we believe this problem is worth exploring and can utilize the information captured in the runtime's dataflow graph.

Another limitation is the need, in some cases, for the end user to provide cost models to their workload. It is difficult to know what constitutes a good cost model, much less an optimal one. With OAs, the end user can at least more easily evaluate different models and their downstream scheduling decisions with little code modification. In this case, the end user can simply change a few parameters and re-run the application, as opposed to re-inserting data transfer statements into the application code for each schedule. However, optimal scheduling remains a complex problem.

OAs are also unable to apply some types of low-level optimizations that require changes to the accelerator library functions. For example, the interface for CUDA streams, a method for overlapping data transfer and compute, is unavailable in cuDF and has a library-specific interface in PyTorch. Because OAs rely on diverse, existing libraries to provide optimized kernel implementations, indirectly calling into lower-level CUDA libraries like cuBLAS, cuDNN, and Thrust, they cannot coordinate calls to interfaces such as CUDA streams across these libraries. Nonetheless, users combining these accelerator libraries by hand would face the same limitation.

10 Related Work

OAs build on split annotations [23], which provide per-function annotations over existing CPU-based libraries to enable cross-function data pipelining and improved cache utilization. OAs extend split annotations by considering several new problems unique to accelerators, including data transfer and allocation across devices, memory limits of accelerators, and the problem of scheduling computations and transfers across CPUs and accelerators. These problems were not considered in the design of split annotations, so they require both an extended annotation interface (§4) and a different runtime and scheduler (§5).

Existing accelerator libraries such as PyTorch [24], cuDF [7], RAPIDS [9], and cuML [8] provide interfaces for targeting GPUs that intend to mirror CPU library APIs. However, they usually cannot handle data that does not fit in the accelerator memory, only support a subset of their CPU counterparts, and invariably involve application rewrites. OAs are a system designed to bridge these shortcomings, by leveraging new accelerator implementations of library functions but using annotations to automatically page and schedule work across the accelerator and CPU in complex applications that call multiple library functions.

Another popular approach for targeting heterogeneous platforms is compilation, where a compiler generates code (e.g., CUDA) underneath an existing library interface. Several solutions exist for data analytics [20–22, 27, 31, 34] and machine learning [11, 13, 30]. As one example, Numba [2] compiles NumPy code into an intermediate representation (IR), and then to optimized CPU or GPU code. However, compilers trade off good performance for high complexity: they are difficult to implement and to integrate into existing libraries, and the generated code may not match the performance of heavily hand-optimized kernels such as linear algebra functions [23]. In contrast, annotation-based approaches such as split annotations achieved similar speedups to these compilers in many cases with substantially less developer effort (e.g., 10× less code than accelerating functions with Weld in the Mozart evaluation [23]) by leveraging individual, hand-written kernel functions and only optimizing the data movement across them. Like split annotations, OAs propose using expert-written kernels to offload computations rather than attempting to generating code that matches the performance of these kernels.

Several existing systems schedule tasks in a heterogeneous environment [16–19, 26, 32, 33]. Our system primarily aims to provide an interface to bridge existing CPU and GPU code. The algorithms presented in these systems are complementary, and can be used to schedule the task graphs produced by OAs.

Hardware-portable languages like OpenCL [29] provide a common interface to target both CPU and accelerators. However, they require end users to write custom code in these languages, whereas our goal is to leverage optimized CPU and accelerator kernels that have already been written by expert developers and automatically invoke these kernels in an application written using high-level APIs.

11 Conclusion

We have presented *offload annotations (OAs)*, a new approach for bridging existing CPU libraries with emerging GPU libraries with no library code changes. Annotators use OAs to specify an accelerator function for a corresponding CPU function, and to define how inputs to a function can be transferred between devices. Optionally, annotators can also annotate allocation functions and provide cost model estimators that assist the runtime in making scheduling decisions. Our runtime, Bach, uses the information encapsulated in OAs to automatically schedule functions in an end-user application across devices, manage data transfer, and page large datasets. We apply OAs to several existing to several existing CPU libraries and show that they can improve their by up to 1200× and a median of 6.3× by offloading work to a GPU, with little to no code changes. We also show that OAs enable workloads that could previously not fit in GPU memory to reap the benefits of hardware acceleration, without manual effort by the application developer.

12 Acknowledgments

We thank our shepherd, Michael Ferdman, the anonymous ATC reviewers, and the members of Stanford Future Data for their invaluable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Black Scholes Formula. http://gosmej1977.blogspot.com/2013/02/black-and-scholes-formula.html, 2013.

[2] Numba. https://numba.pydata.org, 2018.

[3] A Beginner's Guide to Optimizing Pandas Code for Speed. goo.gl/dqwmrG, 2019.

[4] Computational Formulas. https://oag.ca.gov/sites/all/files/agweb/pdfs/cjsc/prof10/formulas.pdf, 2020.

[5] Demo of DBSCAN clustering algorithm. https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py, 2020.

[6] Importance of Feature Scaling. https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html, 2020.

[7] NVIDIA cuDF. https://github.com/rapidsai/cudf, 2020.

[8] NVIDIA cuML. https://github.com/rapidsai/cuml, 2020.

[9] NVIDIA RAPIDS. https://developer.nvidia.com/rapids, 2020.

[10] Truncated Singular Value Decomposition (TSVD). https://github.com/rapidsai/notebooks/blob/branch-0.12/cuml/tsvd_demo.ipynb, 2020.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[12] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *European conference on Object-oriented programming*, pages 428–452. Springer, 2005.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[14] Tianyi David Han and Tarek S Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.

[15] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive KD Tree GPU Ray Tracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 167–174. ACM, 2007.

[16] Víctor J Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 19–33. Springer, 2009.

[17] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T Lewis, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162. IEEE, 2014.

[18] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[19] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 341–352. ACM, 2012.

[20] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, 2011.

[21] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment*, 11(9):1002–1015, 2018.

[22] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[23] Shoumik Palkar and Matei Zaharia. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 291–305, 2019.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[25] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[26] Vignesh T Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 140–147. IEEE, 2012.

[27] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. pages 49–68. ACM, 2013.

[28] Mark Silberstein. GPUs: High-Performance Accelerators for Parallel Applications: The Multicore Transformation (Ubiquity Symposium). *Ubiquity*, 2014(August), August 2014.

[29] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[30] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

[31] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[32] Lei Wang, Yong-zhong Huang, Xin Chen, and Chun-yan Zhang. Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environments. In *2008 International Conference on Computer Science and Information Technology*, pages 228–232. IEEE, 2008.

[33] Yuan Wen, Zheng Wang, and Michael FP O'boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[34] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, volume 8, pages 1–14, 2008.

HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism

Jay H. Park[1], Gyeongchan Yun[1], Chang M. Yi[1], Nguyen T. Nguyen[1], Seungmin Lee[1], Jaesik Choi[2], Sam H. Noh[1], and Young-ri Choi[1]

[1]*UNIST* [2]*KAIST*

Abstract

Deep Neural Network (DNN) models have continuously been growing in size in order to improve the accuracy and quality of the models. Moreover, for training of large DNN models, the use of heterogeneous GPUs is inevitable due to the short release cycle of new GPU architectures. In this paper, we investigate how to enable training of large DNN models on a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training. We present a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates pipelined model parallelism (PMP) with data parallelism (DP). In HetPipe, a group of multiple GPUs, called a *virtual worker*, processes minibatches in a pipelined manner, and multiple such virtual workers employ data parallelism for higher performance. We also propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP) to accommodate both PMP and DP for virtual workers, and provide convergence proof of WSP. Our experimental results on a given heterogeneous setting show that with HetPipe, DNN models converge up to 49% faster compared to the state-of-the-art DP technique.

1 Introduction

Deep Neural Networks have been popularly used to solve various problems such as image classification [16,29], speech recognition [17], topic modeling [3], and text processing [10]. The size of DNN models (i.e., the number of parameters) have continuously been increasing in order to improve the accuracy and quality of models and to deal with complex features of data [19,47,54,55]. The size of input data and batches used for training have also increased to achieve higher accuracy and throughput [19,26].

For training large DNN models, data parallelism [4, 31, 32, 50], which employs multiple workers using parameter servers or AllReduce communication, and model parallelism [12,28,30], which divides the network layers of a DNN model into multiple partitions and assigns each partition to a different GPU, have commonly been leveraged. Furthermore, to mitigate the critical issue of low GPU utilization of naive model parallelism, pipelined model parallelism, where minibatches are continuously fed to the GPUs one after the other and processed in a pipelined manner, has recently been proposed [19,38].

Table 1: Heterogeneous GPUs

	Year	Archi.	CUDA Core	Boost Clock (MHz)	Memory Size (GB)	Memory BW (GB/sec)
TITAN V	2017	Volta	5120	1455	12	653
TITAN RTX	2018	Turing	4608	1770	24	672
GeForce RTX 2060	2019	Turing	1920	1680	6	336
Quadro P4000	2017	Pascal	1792	1480	8	243

For training DNN models, the use of GPU clusters is now commonplace. In such an environment, the use of heterogeneous GPUs is inevitable due to the short release cycle of new GPU architectures [24]. Moreover, several types of GPUs targeted for high-end servers, workstations, and desktops are being released for purchase [39–42]. Due to their cost-effectiveness, less expensive GPUs targeted for desktops and workstations, rather than high-end servers are also commonly used for machine learning training, especially for small and medium size clusters [14,21,49,56,57,59]. Due to the same reason, spot instances with different types of GPUs that are offered by cloud service providers are being used [2,24,36]. Table 1 shows the hardware specifications for four different types of GPUs, along with their market release years, that we have purchased in our institution in the short span of the last three years. Each, at the time of purchase, was (close to) state-of-the-art affordable with what budget we could muster. With technology advancing in such rapid pace, these systems have become outdated. Some of the systems have become old technologies that, individually, are unable to run large DNN models that are common today. Such situations with clusters of heterogeneous GPUs should now be commonplace.

There are benefits to enabling DNN training with heterogeneous resources. First, it allows for large model training with lower-class GPUs. While unable to train individually due to their limited resources, aggregated together, they may be used for training. These GPUs, which likely would have been retired, become usable, possibly used to create (virtual) workers that show similar performance as high-class GPUs. Second, low-class GPUs can be used to improve the performance of even high-class GPUs by incrementally adding on the resources of the (old) lower class systems to the (new) high-class systems. We call a group of aggregated GPUs that could satisfy the resource constraint and be used for training a *virtual worker*. Internally, such a virtual worker could leverage pipelined model parallelism (PMP) to process a minibatch, while externally, a number of virtual workers could leverage data parallelism (DP) for higher performance.

In this paper, we explore the integration of PMP and DP to maximize the parallelism of DNN model training. In particular, we investigate a DNN model training system, which employs both PMP and DP, for a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training large models. Integrating DP to PMP may sound trivial, but in fact, it is quite challenging. In this setting, each virtual worker is continuously processing multiple minibatches in a pipelined manner and thus, all the virtual workers can be in different states. Thus, the key question here is, what weight version should be used by each virtual worker to synchronize with other virtual workers? Numerous questions need to be answered to answer this question: 1) How many new minibatches can start being processed while waiting for global updates from the parameter server? 2) Can synchronization occur at any point of processing the minibatches? 3) How can convergence be guaranteed when such synchronization occurs? 4) What version of parameters is used for the next minibatch while previous minibatches are still executing within each virtual worker? (This question is also considered to some extent in a prior work [38].) And so on. Furthermore, there are also many challenges that need to be overcome to ideally leverage a heterogeneous GPU cluster for DNN training: How are the heterogeneous GPUs to be divided and allocated into virtual workers? How do we reduce virtual worker stragglers when we consider DP? How do we partition the model to maximize the performance of PMP using heterogeneous GPUs?

While DP [4, 31, 32, 50], PMP [19, 38], and heterogeneity [24, 25, 33] for training have been considered separately, to the best of our knowledge, this is the first paper that tackles these issues together in attempting to answer *some* of the aforementioned questions. In this work, we design a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates PMP of a virtual worker, which is composed of multiple (possibly whimpy) heterogeneous GPUs, with DP of virtual workers using parameter servers to enable and also speed up training of large models. HetPipe can aggregate heterogeneous resources from multiple GPUs to form a virtual worker such that the performance of each virtual worker is similar to each other, reducing the straggler problem. For HetPipe, we propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP). WSP is adapted from the Stale Synchronous Parallel (SSP) model [18] to accommodate both PMP and DP for multiple virtual workers unlike existing synchronization models. We also prove the convergence of WSP. Note that while HetPipe would work in a homogeneous GPU cluster in training a large model that cannot be loaded into the memory of a single GPU, with the rapid turnaround of newer GPU architectures, it is more likely that one will end up with a cluster of heterogeneous GPUs. This is the environment that we target.

We implement HetPipe by modifying TensorFlow, a commonly used machine learning training system. We evaluate the performance of HetPipe for two DNN models using a heterogeneous GPU cluster composed of four different types of GPUs. Our experimental results demonstrate that the performance of HetPipe is better than that of the state-of-the-art DP via Horovod [50] that uses AllReduce communication [45]. This is because HetPipe mitigates the straggler problem, and also because it enables each virtual worker and the parameter server to intra-communicate for all parameter updates, significantly reducing communication overhead. Compared to Horovod, the convergence of VGG-19 with a large parameter set to a desired accuracy becomes 49% faster, and that of ResNet-152 which is too big to be loaded in four whimpy GPUs in our cluster becomes 39% faster by using all the GPUs (including whimpy ones).

Strategies to leverage PMP have been explored in previous studies [7, 19, 27, 38]. Compared to these, our study makes forward strides in three aspects. First, we generalize PMP of a virtual worker to be used together with DP of virtual workers, increasing the parallelism of DNN model training. Consequently, this results in speeding up training. Second, we consider a heterogeneous GPU cluster, which allows the use of GPUs, which otherwise, could not be used for training. Finally, we present a parameter synchronization model that guarantees convergence, of which we provide a proof, for training models using PMP with DP. We provide a more in-depth comparative discussion on these studies in Section 2.2.

2 Background

2.1 Data Parallelism

Training of a DNN model is processed by a *forward pass* followed by a *backward pass* for each *minibatch*, which is a subset of training samples, in a popularly used *stochastic gradient descent* (SGD) method. For each minibatch, the weight updates, i.e., *gradients*, are computed to update weights (or parameters) w of the model.

Data parallelism (DP) utilizes multiple workers to speed up training of a DNN model. It divides the training dataset into subsets and assigns each worker a different subset. Each worker has a replica of the DNN model and processes each minibatch in the subset, thereby computing the weight updates. Therefore, if a DNN model cannot be loaded into the memory of a single GPU, DP cannot be used.

Among the multiple workers, the parameters are synchronized using parameter servers [31] or AllReduce communications [32, 50]. For *Bulk Synchronous Parallel* (BSP) [1, 35], each worker must wait for all other workers to finish the current minibatch p before it starts to process the next minibatch $p+1$ so that it can use an updated version of the weights for minibatch $p+1$. For *Asynchronous Parallel* (ASP) [1, 48], each worker need not wait for other workers to finish minibatch p, possibly using a stale version of the weights. With BSP, which is possible for both the parameter servers and

AllReduce communications, the system may suffer from high synchronization overhead, especially in a heterogeneous GPU cluster where each worker with a different GPU provides different training performance [33]. On the other hand, while ASP, which is possible for the parameter servers, has no synchronization overhead, it is known that ASP does not ensure convergence [48, 58].

A method that takes the middle ground between BSP and ASP is *Stale Synchronous Parallel* (SSP) [18]. With SSP, each worker is allowed to proceed the training of minibatches using a *stale* version of the weights that may not reflect the most recent updates computed by other workers. Thus, workers need not synchronize with other workers whenever it finishes the processing of a minibatch. As such, parameter staleness can occur. However, this staleness is bounded as defined by the user and referred to as the *staleness threshold*. As SSP is beneficial when worker performance is varied, it has been explored especially in the context of heterogeneous systems [24].

In SSP, each worker periodically pushes the weight updates to the parameter server. This synchronization interval is called a *clock*. Thus, each worker increases its local clock by one for every iteration, which is the training period of a minibatch. For a given staleness threshold s where $s \geq 0$, each worker with clock c is allowed to use a stale version of the weights, which includes all the updates from iteration 0 to $c-s-1$ and, possibly, more recent updates past iteration $c-s-1$. That is, a worker can continue training of the next minibatch with parameters whose updates may be missing from up to the s most recent minibatches.

2.2 Model Parallelism and Pipeline Execution

Model parallelism (MP) is typically exploited for large DNN models that are too large to be loaded into memory of a single GPU. In particular, a DNN model composed of multiple layers is divided into k partitions and each partition is assigned to a different GPU. Each GPU executes both the forward and backward passes for the layers of the assigned partition. *Note that it is important to execute the forward and backward passes of a partition on the same GPU* as the activation result computed for the minibatch during the forward pass needs to be kept in the GPU memory until the backward pass of the same minibatch for efficient convergence, as similarly discussed by Narayanan and others [38]. Otherwise, considerable extra overhead will incur for managing the activation through either recomputation or memory management.

In the basic form of MP, k GPUs, individually, act as one *virtual worker* to process a minibatch as follows: For each minibatch, execution of the forward pass starts from GPU$_1$ up to GPU$_k$. When each GPU$_i$, where $1 \leq i < k$, completes the forward pass of the assigned partition, it sends the computed activations of *only the last layer in its partition* to GPU$_{i+1}$. Once GPU$_k$ finishes the forward pass of its partition, the backward pass of the minibatch is executed from GPU$_k$ down to GPU$_1$. When each GPU$_{i'}$, where $1 < i' \leq k$, finishes the backward pass, it sends the computed local gradients of *only the first layer in its assigned partition* to GPU$_{i'-1}$. This basic form of MP results in low GPU utilization as only one GPU is actively executing either the forward or backward pass. Nonetheless, MP allows execution of large DNN models that are too large for a single GPU.

Table 2: Comparison of HetPipe with GPipe and PipeDream

	GPipe	PipeDream	HetPipe
Heterogeneous Cluster Support	No	No	Yes
Target Large Model Training	Yes	No	Yes
Number of (Virtual) Workers	1	1	N
Data Parallelism	Extensible	Partition	Virtual Workers
Proof of Convergence	Analytical	Empirical	Analytical

To improve utilization of the GPUs in a virtual worker, minibatches can be processed in a pipelined manner. The subsequent minibatches are fed into the first GPU in MP (i.e., GPU$_1$) one by one once the GPU completes the processing of the previous minibatch. This allows for multiple GPUs to simultaneously execute either the forward or backward pass of their assigned layers for different minibatches. This is referred to as Pipelined Model Parallelism (PMP).

This PMP strategy has been investigated in previous studies [19, 38]. PipeDream exploits PMP of a single virtual worker to avoid the parameter communication overhead of DP [38]. Considering only homogeneous GPUs, when PipeDream partitions a model into stages to maximize pipeline performance, it does not take into account the memory requirement of a GPU that depends on the stage of a pipeline. Thus, PipeDream processes a limited number of minibatches, which is large enough to saturate the pipeline, to reduce memory overhead. PipeDream also provides a form of DP, but it considers DP within a virtual worker to speed up the execution of lagging layers. No proof of single pipeline convergence is provided in PipeDream. Note that without a parameter synchronization model such as WSP, it is not possible to properly run DP over multiple PipeDream virtual workers via parameter servers or AllReduce communication.

GPipe is a scheme that leverages PMP of a single virtual worker to support large DNN models, also in a homogeneous GPU cluster [19]. In GPipe, a minibatch is divided into multiple *microbatches* that are injected into the pipeline. Using the same weights, GPipe executes the forward passes for all the microbatches, and then executes the backward passes for them. When the backward pass of the last microbatch is done, it updates the weights all together for the minibatch. GPipe incurs frequent pipeline flushes, possibly resulting in low GPU utilization [38]. In GPipe, DP of multiple virtual workers can be done using existing synchronization schemes like BSP as a virtual worker processes one minibatch at a time. GPipe saves on GPU memory by recomputing the activations again in the backward pass instead of keeping the activations computed in the forward pass in memory. We do not use this optimization though there are no fundamental reasons forbidding it. A comparison of HetPipe with previous studies is given in Table 2.

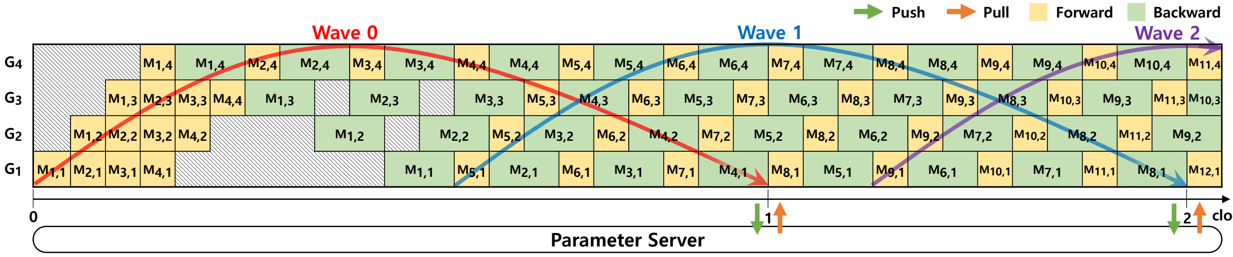


Figure 1: Pipeline execution of minibatches where $M_{p,k}$ indicates the execution of a minibatch p in partition k, which is executed in GPU$_k$ and the yellow and green colors indicate the forward and backward passes, respectively.

3 System Overview

The system that we propose focuses on training a large DNN model in a *heterogeneous GPU cluster* composed of various types of GPUs that have different computation capability and memory capacity. In such settings, for some types of GPUs in the cluster, the DNN model of interest may be too large to be loaded into the memory of a single GPU. The system that we propose in this paper leverages both pipelined model parallelism (PMP) and data parallelism (DP) to enable training of such large DNN models and, in the process, enhance performance as well as the utilization of the heterogeneous GPU resources of the cluster.

Figure 2 shows the architecture of the proposed cluster system composed of H nodes. Each node comprises a homogeneous set of GPUs, but the GPUs (and memory capacity) of the nodes themselves can be heterogeneous. Two key novelties exist in this architecture. First, DP is supported through a notion of a *virtual worker* (VW), which consists of k, possibly heterogeneous, GPUs, and encapsulates the notion of a worker in typical DNN systems. That is, a virtual worker is used to train the DNN model. In Figure 2, note that there are N virtual workers with 4 GPUs each, that is, $k = 4$, and that the GPUs comprising the virtual worker may be different for each virtual worker. While in this paper we consider k to be constant for each virtual worker, our design does not restrain it to be so; this is simply a choice we make for simplicity. The key aspect here is that a virtual worker allows DP by aggregating GPUs possibly even when individual GPUs may be resource limited.

The second novelty is that each virtual worker processes each minibatch based on model parallelism, in a pipelined manner, to fully utilize the GPU resources, as shown in Figure 1, to accommodate large DNN models. While PMP has been proposed before (which we compare in Section 2.2), to the best of our knowledge, we are the first to present PMP in a heterogeneous setting. We refer to our system as *HetPipe* as it is *het*erogeneous, in GPUs, across and, possibly, within virtual workers and makes use of *pipe*lining in virtual workers for resource efficiency.

To train DNN models based on pipelined model parallelism in virtual workers, the *resource allocator* first assigns k GPUs to each virtual worker based on a resource allocation policy (which will be discussed in Section 8.1). Note that for allocating the heterogeneous GPUs to the virtual workers, the resource allocation policy must consider several factors such as the performance of individual GPUs as well as the communication overhead caused by sending activations and gradients within a virtual worker, and synchronizing the weights among the virtual workers and the parameter server. Then, for the given DNN model and allocated k GPUs, the *model partitioner* divides the model into k partitions for the virtual worker such that the performance of the pipeline executed in the virtual worker can be maximized.

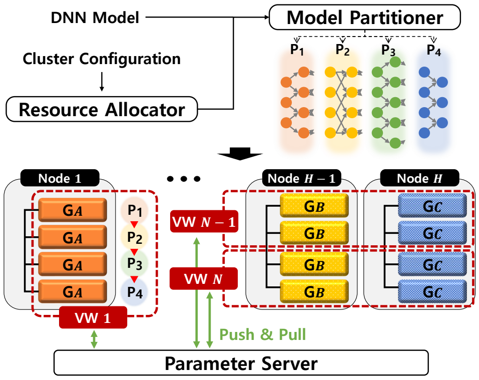


Figure 2: System architecture (VW: Virtual Worker)

As any typical DP, multiple virtual workers must periodically synchronize the global parameters via parameter servers or AllReduce communication; in HetPipe, parameter servers are used to maintain the global weights. Each virtual worker has a local copy of the global weights and periodically synchronizes the weights with the parameter server. Evidently, when managing the weights within a virtual worker and across virtual workers, two types of staleness, *local staleness* and *global staleness*, need to be permitted to improve the performance of DNN training. Local staleness refers to staleness within a virtual worker. As each virtual worker processes minibatches in a pipelined manner, there are multiple minibatches that are being processed in parallel. Thus, staleness is inevitable as weights seen by a minibatch may not reflect the updates of all of its previous minibatches.

Global staleness, on the other hand, is similar to the staleness notion introduced by Ho et al. [18]. That is, the system needs to reduce communication overhead between the parameter server and (virtual) workers, and, in our case, also mitigate the synchronization overhead caused by possibly heterogeneous virtual workers. Therefore, similarly to SSP [18], each virtual worker should be allowed to proceed training without querying the global weights for every minibatch, unless its local copy is so old such that there are too many missing recent updates made by other virtual workers. Note that such staleness condition is set by the user [18].

For our system, we propose the *Wave Synchronous Parallel* (WSP) model to synchronize the weights. A *wave* is a sequence of minibatches that are processed concurrently in a virtual worker. Let the number of minibatches in a wave be N_m. Within a wave, processing of the i-th minibatch is allowed to proceed without waiting for the preceding minibatchs i' to be completed, where $1 < i \leq N_m$ and $1 \leq i' < i$. That is, there is no dependency among the weights used by minibatches in the same wave. As the virtual worker does not enforce the updates even from the first minibatch in a wave to be reflected in the weights used by the last minibatch, the local staleness threshold in WSP is $N_m - 1$. Moreover, *each virtual worker only pushes the aggregated updates from all the minibatches in a wave, instead of for every minibatch, to the parameter server.* This results in considerable reduction in communication overhead.

As it is important that the results generated through our proposed system configuration are correct [18, 24, 60], we show the convergence of our methodology in Section 6.

Note that HetPipe uses parameter servers, which may incur synchronization and communication overhead. However, HetPipe mitigates such overhead by permitting global staleness among virtual workers and executing the pipeline in each virtual worker such that it continues to process minibatches that have already been injected while waiting for the parameter update. We believe HetPipe can be further optimized by taking decentralized approaches, but leave this for future work.

4 Pipelined Model Parallelism Within a VW

Number of Minibatches in the Pipeline: In our system, each virtual worker processes up to N_m minibatches concurrently in a pipeline manner so that the executions of the minibatches can overlap. Given a DNN model and k GPUs, the maximum number of minibatches executed concurrently in the virtual worker, Max_m, is basically determined by the memory requirement for training the model. For a model that requires a huge amount of memory for output activations and weights, Max_m may be less than k. Note that in such cases, the utilization of each GPU is unlikely to be high.

N_m, the actual number of minibatches in the pipeline will be $N_m \leq Max_m$ and basically determined by considering the throughput of the pipeline. Note that N_m must be the same in every virtual worker, and thus, N_m is set to the minimum Max_m among all the virtual workers. N_m will affect the local staleness that we discuss later in this section.

Model Partitioning: To train a DNN model, a set of k GPUs is allocated to a virtual worker by a resource allocation policy, which we discuss in Section 8.1. For now, let us assume that k, the number of possibly heterogeneous GPUs, and N_m are given. Then, a partitioning algorithm is employed to divide multiple layers of the model into k partitions, assigning them to the k different GPUs. The goal of the partitioning algorithm is to maximize the performance of the pipeline, while satisfying the memory requirement of each partition to process N_m minibatches.

In particular, in this study, for memory, we consider the fact that the actual memory requirement will vary depending on the stage of the pipeline that the GPU is used for. For example, contrast GPU$_4$ and GPU$_1$ in Figure 1. GPU$_4$, the GPU that handles the last stage of the pipeline, handles only one minibatch at a time and is immediately done with the minibatch as exemplified by the yellow (forward pass) and green (backward pass) M$_{i,4}$ pairs for $i = 1, 2, ...,$ that are side-by-side. In contrast, for GPU$_1$, the yellow and green M$_{i,1}$ pairs are far apart, meaning that the forward pass M$_{i,1}$ needs to hold up memory until the backward pass M$_{i,1}$ is finished with its execution. Thus, with GPU$_1$, the memory requirement is high as it needs to hold on to the results of the forward pass for all stages of the pipeline. This variance in memory requirement is considered in partitioning the layers.

Execution time must also be considered when partitioning the layers. To do so, we calculate the execution time of a partition to be the sum of the computation time of all the layers in the partition and the communication time needed for receiving the activations (in the forward pass) and local gradients (in the backward pass). Our partitioning algorithm attempts to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement.

Partition Scheduling: Once the partition is set, the partitions need to be scheduled for each of the GPUs. Each GPU$_q$ responsible for partition q may have multiple forward pass and backward pass tasks to schedule at a time. Each GPU schedules a task by enforcing the following conditions:

1. A forward pass task for a minibatch p will be executed only after a forward pass task for every minibatch p' is done where $1 \leq p' < p$.
2. Similarly, a backward pass task for a minibatch p will be executed only after a backward pass task for every minibatch p' is done where $1 \leq p' < p$.
3. Among multiple forward and backward pass tasks, a FIFO scheduling policy is used.

Note that in the last partition, for a minibatch, processing a forward pass immediately followed by a backward pass is executed as a single task.

Considering Staleness: Given the description of pipelining, the question of staleness of weights used needs to be considered. That is, as a minibatch is scheduled, it may be that the layers are not using the most up-to-date weights. For example, in Figure 1, when the forward pass M$_{2,1}$, the second minibatch, begins to be processed, it must use stale weights as the first minibatch has not completed and hence, the changes in the weights due to the first minibatch have not yet been appropriately reflected, which is in contrast with typical processing where minibatches are processed one at a time. We now discuss how this staleness issue is considered.

Let *local staleness* be the maximum number of missing updates from the most recent minibatches that is allowed for a minibatch to proceed in a virtual worker. As training with N_m minibatches can proceed in parallel in a virtual worker, the local staleness threshold, s_{local}, is determined as $N_m - 1$, where $1 \leq N_m \leq Max_m$. If $N_m = 1$, the behavior is exactly the same as naive model parallelism. Larger N_m may improve the performance (i.e., throughput) of the pipeline as a larger number of concurrent minibatches are executed, but local staleness increases, possibly affecting the convergence of training. In a real setting, typically, N_m will not be large enough to affect convergence as it will be bounded by the total amount of GPU memory of a virtual worker.

Such local staleness also exists in PipeDream [38]. As PipeDream basically employs weight stashing that uses the latest version of weights available on each partition to execute the forward pass of a minibatch, a different version of weights is used across partitions for the same minibatch. Unfortunately, PipeDream only shows empirical evidence of convergence when weight stashing is used. Note that PipeDream also discusses vertical sync, which is similar to HetPipe, but it excludes vertical sync in its evaluations [38].

Now let w_p be the weights used by minibatch p. Then, initially, we can assume that w_0, the initial version of weights, is given to the virtual worker. Then, the first $(s_{local} + 1)$ minibatches are processed in a pipelined manner with $w_0 = w_1 = \cdots = w_{s_{local}} = w_{s_{local}+1}$.

To accommodate staleness in our system, when processing of minibatch p completes, the virtual worker updates the local version of the weights, w_{local} as $w_{local} = w_{local} + u_p$, where u_p is the updates computed by processing minibatch p. Therefore, in HetPipe, weights are not updated layer by layer and w_{local} is a consistent version of weights across partitions. When the virtual worker starts to process a new minibatch, it makes use of the latest value of w_{local} without waiting for the other minibatches to update their weights. For example, once the virtual worker is done for minibatch 1 and updates w_{local} with u_1, it will start to process minibatch $s_{local} + 2$ by using the updated weights without waiting for minibatches 2 up to $s_{local} + 1$ to be completed. Similarly, when the virtual worker is done with minibatch $s_{local} + 1$ and updates w_{local} with $u_{s_{local}+1}$, it will start to process minibatch $2 \times (s_{local} + 1)$ without waiting for the previous most recent s_{local} minibatches to be completed. Therefore, except for the initial minibatches 1 to $s_{local} + 1$, for minibatch p the virtual worker will use the version of the weights that reflects (at least) all the local updates from minibatches 1 to $p - (s_{local} + 1)$. Note that for every minibatch p, w_p must be kept in GPU memory until the backward pass for p is executed.

Note that staleness in SSP is caused by the different processing speed of minibatches among multiple workers. Thus, in SSP, staleness is used as a means to reduce the synchronization and communication overhead. However, local staleness in HetPipe is caused inherently as minibatches are processed in a pipelined manner within a virtual worker.

5 Data Parallelism with Multiple VWs

In this section, we discuss data parallelism (DP) with virtual workers. The first and foremost observation of DP being supported with virtual workers is that the virtual workers may be composed of (whimpy) heterogeneous GPUs. While it is well known that DP helps expedite DNN execution, DP, in typical systems, is not possible if individual GPUs, that is, workers, do not have sufficient resources to handle the DNN model, in particular, large DNNs. By allowing a virtual worker to be composed of multiple GPUs that are lacking in resources, our system allows DP even with whimpy GPUs. The other key observation in properly supporting DP with virtual workers is that each virtual worker now retains local staleness as discussed in Section 4. Making sure that, despite such individual staleness, we understand and show that the results obtained from DP among virtual workers (globally) converge is an important issue that must be addressed. The rest of the section elaborates on this matter.

Workings of WSP: As stated in the system overview, HetPipe uses parameter servers. We assume that such synchronization occurs in *clock* units, a notion taken from SSP [18]. Precisely, a clock unit is defined as the progress of completing one wave. Recall from Section 3 (and Figure 1) that a wave is a sequence of $s_{local} + 1$ minibatches concurrently executed such that a virtual worker is allowed to process a later minibatch in a wave without updates from an earlier minibatch in the same wave.

Similarly to SSP (which, however, considers the staleness of weights only in DP), each virtual worker maintains a local clock c_{local}, while the parameter server maintains a global clock c_{global}, which holds the minimum c_{local} value of all the virtual workers. Initially, the local clocks and the global clock are 0. At the end of every clock c, each virtual worker completes the execution of all the minibatches in wave c. At this point, the virtual worker computes the aggregated updates from minibatch $c \times (s_{local} + 1) + 1$ to minibatch $(c+1) \times (s_{local} + 1)$ and pushes the updates $\tilde{u}$ to the parameter server. We see that, similar to in SSP [18], $\tilde{u}$ is synchronized with a clock value c. For example, as shown in Figure 1 where $s_{local} = 3$, at the end of clock 0, the virtual worker pushes the aggregated updates of wave 0, which is composed of minibatches from 1 to 4, and at the end of clock 1, the aggregated updates of wave 1, which is composed of minibatches from 5 to 8, and so on. It is important to note that in WSP, the virtual worker pushes $\tilde{u}$ to the parameter server for every wave, instead of pushing $\tilde{u}$ for every minibatch, which will significantly reduce the communication overhead.

When the parameter server receives the updates ũ from the virtual worker, the parameter server updates the global version of the weights as $w_{global} = w_{global} +$ ũ. Note that the parameter server updates its c_{global} to $c+1$ only after every virtual worker has pushed the aggregated updates of wave c.

In WSP, each virtual worker is allowed to proceed training without retrieving the global weights for every wave. Thus, the virtual worker may use a weight version that, from a global standpoint, may be stale, as the most recent updates received by the parameter servers may not be reflected in its local version of the weights. We discuss how global staleness among the virtual workers is bounded.

Global Staleness Bound: Let *clock distance* be the difference in c_{local} between the fastest and slowest virtual workers in the system. Therefore, a virtual worker with local clock c, where $c \geq D+1$, must use a version of the weights that includes all the (aggregated) updates from wave 0 up to $c-D-1$. Also, the weight version may include some recent global updates from other virtual workers and some recent local updates within the virtual worker beyond wave $c-D-1$.

When a virtual worker pulls the global weights at the end of clock c to maintain this distance, it may need to wait for other virtual workers to push their updates upon completion of wave $c-D$. However, while a virtual worker waits for other virtual workers to possibly catch up at the end of clock c, local processing is allowed to proceed with s_{local} minibatches of wave $c+1$ as the minibatches are executed in a pipelined manner. Take, for example, the case when $D=0$ and $s_{local}=3$ in Figure 3 (and Figure 1). As a virtual worker, VW1, completes minibatch 4, it computes the aggregated updates ũ for wave 0 (composed of minibatches 1 to 4) and pushes ũ to the parameter server. VW1 now waits for the other virtual workers to complete wave 0 before proceeding with minibatch 8. However, note that as shown in the figure, VW1 has already started to process minibatches 5, 6 and 7, which belong to wave 1, while its local clock is still 0. Similarly, once it completes minibatch 8, it pushes the aggregated updates ũ for wave 1 (composed of minibatches 5 to 8) to the parameter server; in the meantime, it has already started processing minibatches 9, 10, and 11, which belong to wave 2, while its clock is still 1.

Note that this processing of local minibatches in the virtual worker does not violate the local staleness bound. Note also that when $D=0$, each virtual worker must wait for each other at the end of every clock to synchronize the weights for every wave, which is BSP-like behavior with pipelined execution in each virtual worker.

Now let us define the *global staleness bound*, s_{global}, to be the maximum number of missing updates from the most recent minibatches, *globally computed by all the other virtual workers in the system*, that is allowed for a minibatch to proceed in a virtual worker. We want to identify s_{global} based on our discussion so far. This will allow each virtual worker to determine whether it can proceed with its current minibatch.

Initially, all virtual workers start processing the first $(D+1)$ waves without querying the global weights from the parameter server. Furthermore, they can start to process up to s_{local} minibatches of the next wave before receiving the global weights that include the recent updates as discussed above. Therefore, for those initial minibatches, the virtual worker uses w_0 or a weight version that may include some recent local updates.

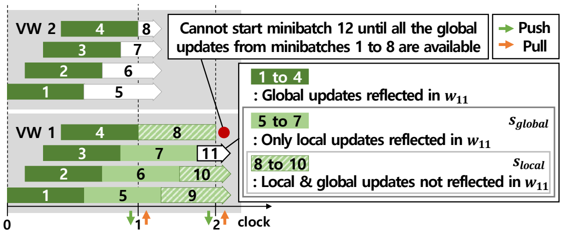

VW 2
4
8
3
7
2
6
1
5
VW 1
4
8
3
7
11
2
6
10
1
5
9
0
1
2
clock
Cannot start minibatch 12 until all the global updates from minibatches 1 to 8 are available
Push
Pull
1 to 4
: Global updates reflected in w_{11}
5 to 7
s_{global}
: Only local updates reflected in w_{11}
8 to 10
s_{local}
: Local & global updates not reflected in w_{11}

Figure 3: Local and global staleness with WSP

For any minibatch p thereafter, that is, where $p > (D+1) \times (s_{local}+1) + s_{local}$, p must use a weight version that reflects, at the very least, all the global updates from all the other virtual workers from minibatch 1 to minibatch $p - (s_{global}+1)$, where $s_{global} = (D+1) \times (s_{local}+1) + s_{local} - 1$. The first term of this equation is due to the fact that a virtual worker is allowed to proceed with the next $(D+1)$ waves (i.e., $(D+1) \times (s_{local}+1)$ minibatches), and the second term is due to the additional s_{local} minibatches that can be started because of pipelined execution. Continuing with the example in Figure 3, where $D=0$ and $s_{local}=3$, VW1 proceeds the training of minibatch 11 without the global and/or local updates from wave 1 (minibatches 5 to 8) or the two local updates from minibatches 9 and 10 (i.e., having $s_{global}=6$). Thus, it must have a version of the weights that includes all the global updates from minibatches 1 to 4. Actually, the weight version used for minibatch 11 includes three local updates from minibatches 5, 6, and 7, along with all the global updates from wave 0. In case of minibatch 12, it cannot start the training until global updates up to minibatch 8 are received.

6 Convergence Analysis

In this section, we discuss the convergence property of the WSP model. Let N be the number of virtual workers and $u_{n,p}$ be the update of worker n at minibatch execution p. Let $s_g = s_{global}$ and $s_l = s_{local}+1$, and following the analysis of [18], the noisy weight parameter[1] $\tilde{w}_{n,p}$ for worker n at minibatch execution p, is decomposed into

$$\tilde{w}_{n,p} = w_0 + \left[\sum_{n'=1}^{N} \sum_{p'=1}^{p-s_g-1} u_{n',p'}\right] + \left[\sum_{p' \in \mathcal{C}_{n,p}} u_{n,p'}\right] + \left[\sum_{(n',p') \in \mathcal{E}_{n,p}} u_{n',p'}\right]. \quad (1)$$

[1] In this section, we use the term 'weight parameter' to denote all weights of a network. Thus, the weight parameters refer to a set of weights of networks.

Here w_0 refers to the initial parameter, and the noisy weight parameter has three other terms which respectively include

1. updates of all workers (guaranteed to be included) to process minibatch execution p,
2. $\mathcal{C}_{n,p} \subseteq [p-s_g, p-1]$: the index set of the latest updates of the querying worker n in the range of current global staleness bound, and
3. $\mathcal{E}_{n,p} \subseteq ([1,N] \backslash \{n\}) \times [p-s_g, p+s_g+s_l]$: the index set of extra updates of other workers in the range of the current global staleness bound. When execution p is not at a synchronization point, $\mathcal{E}_{n,p} = \emptyset$.

We define $\{u_t\}$ as the sequence of updates of each virtual worker after processing each minibatch and $\{w_t = w_0 + \sum_{t'=0}^{t-s_l N} u_{t'}\}$ as the reference sequence of weight parameters, where $u_t := u_{t \bmod N, \lfloor t/N \rfloor + t \bmod s_l}$, in which we loop over the workers ($t \bmod N$) and over each update after a minibatch execution inside a worker ($\lfloor t/N \rfloor + t \bmod s_l$). Here, $s_l N$ ($= s_l \times N$) is the number of total minibatch updates in one wave from all virtual workers. Since a virtual worker uses a version of the weight parameter that reflects all the local updates from minibatch 1 to $p - s_l$ for worker p, the reference and noisy sequences at iteration t are updated up to $t - s_l N$. The set $\mathcal{E}_t$ and the noisy weight parameter $\tilde{w}_t$ are defined similarly and the difference between w_t and $\tilde{w}_t$ is $\tilde{w}_t = w_t - \left[\sum_{i \in \mathcal{R}_t} u_i\right] + \left[\sum_{i \in \mathcal{Q}_t} u_i\right]$ where $\mathcal{R}_t$ is the index set of missing updates in the reference weight parameter but not in noisy weight parameter, and $\mathcal{Q}_t$ is the index set of extra updates in the noisy weight parameter but not in reference weight parameter.

After T updates, we represent the target function as $f(w) := \frac{1}{T}\sum_{t=1}^{T} f_t(w)$, the regret of two functions with $\tilde{w}_t$, the parameter learned from the noisy update, and w^*, the parameter learned from the synchronized update is $R[W] := \frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{w}_t) - f(w^*)$.

Thus, when we bound the regret of the two functions, we can bound the error of the noisy updates incurred by the distributed pipeline staleness gradient descent. We first bound the cardinality of $\mathcal{R}_t$ and $\mathcal{Q}_t$ in the following lemma.

Lemma 1. *The following two inequalities, $|\mathcal{R}_t| + |\mathcal{Q}_t| \leq (2s_g + s_l)(N-1)$ and $\min(\mathcal{R}_t \cup \mathcal{Q}_t) \geq \max(1, t - (s_g + s_l)N)$, hold.*

Proof. Since $\mathcal{Q}_t \subseteq \mathcal{E}_t$ and $\mathcal{R}_t \subseteq \mathcal{E}_t \backslash \mathcal{Q}_t$, $|\mathcal{R}_t| + |\mathcal{Q}_t| \leq |\mathcal{E}_t| \leq (2s_g + s_l)(N-1)$. The second claim follows from $\mathcal{E}_t \supseteq \mathcal{R}_t \cup \mathcal{Q}_t$. □

With the following two assumptions, the proof of convergence generally follows Qirong et al. [18][2]

Assumption 1. ***(L-Lipschitz components)*** *For all t, the component function f_t is convex and has bounded subdifferential $\|\nabla f_t(w)\| \leq L$, in which $L > 0$ is a constant.*

Assumption 2. ***(Bounded distances)*** *For all w, w', the distance between them is bounded $D(w\|w') \leq M$, in which $M > 0$ is a constant.*

We also denote $\frac{1}{2}\|w - w'\|^2$ as $D(w\|w')$. Then, we can bound the regret of the function trained with our noisy distributed, pipeline update as in Theorem 1.

Theorem 1. *Suppose w^* is the minimizer of $f(w)$. Let $u_t := -\eta_t \nabla f_t(\tilde{w}_t)$ where $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{M}{L\sqrt{(2s_g+s_l)N}}$, in which M, L are the constants defined in the assumptions. Then the regret is bounded as $R[W] \leq 4ML\sqrt{\frac{(2s_g+s_l)N}{T}}$.*

Our theoretical results are similar with existing work on non pipelined version of staleness update [18, 24]. However, we reflect the new characteristics of distributed pipeline staleness update in Lemma 1, and thus in Theorem 1.

7 Partitioning Algorithm

Recall that the goal of our partitioning algorithm is to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement. To obtain a performance model to predict the execution time of each layer of a model in a heterogeneous GPU, we first profile the DNN model on each of the different types of GPUs in a cluster, where we measure the computation time of each layer of the model. For GPU memory usage, we measure the usage of each layer (by using the logging feature of TensorFlow) on only one GPU type (as it is roughly the same for all GPU types). For profiling the memory usage on a whimpy node, we measure the memory usage of each layer using a small batch size and then multiply it for the target batch size. To compute the memory requirement for a given partition, we take into account the total memory usage to store the data to process the layers as well as the maximum number of minibatches concurrently assigned to the partition.

For communication time between layers in the model, we first derive the amount of input data for each layer in the forward and backward pass from the model graph. For the given data size, we predict intra-node communication based on the PCI-e bandwidth, then multiply it by a scaling-down constant (which is similarly done in Paleo [46]), since in practice, it is not possible to utilize the peak bandwidth. The scaling-down constant is derived by running a synthetic model that sends various sizes of data from one GPU to another GPU in the same node. For inter-node communication (via InfiniBand), we use linear regression to estimate the communication time for the given data size. To build a prediction model, we collect 27 samples by training two DNN models, used in our experiments, with arbitrary partitions. Note that in this work, the heterogeneity of network performance such as slow network links is not considered (as in [33]). However, for such cases, we can extend our partitioning algorithm to consider different network performance between two nodes when estimating the communication time. Also, a model that estimates the memory requirement for each stage more accurately will be helpful in partitioning a DNN model in a more balanced manner.

[2] The full proof is omitted due to space, but can be found in [44].

To find the best partitions of a DNN model, we make use of CPLEX, which is an optimizer for solving linear programming problems [20]. The memory requirement for each partition on the pipeline to support N_m concurrent minibatches is provided as a constraint to the optimizer. The algorithm will return partitions for a model with a certain batch size only if it finds partitions that meet the memory requirement for the given GPUs. Also, the optimizer checks all the different orders of the given heterogeneous GPUs for a single virtual worker to partition and place layers of the DNN model on them.

8 Experimental Results

8.1 Methodology

Heterogeneous GPU cluster: In our experiments, we use four nodes with two Intel Xeon Octa-core E5-2620 v4 processors (2.10 GHz) connected via InfiniBand (56 Gbps). Each node has 64 GB memory and 4 homogeneous GPUs. Each node is configured with a different type of GPU as shown in Table 1. Thus, the total number of GPUs in our cluster is 16. Each GPU is equipped with PCIe-3×16 (15.75 GB/s). Ubuntu 16.04 LTS with Linux kernel version 4.4 is used. We implement HetPipe based on the WSP model by modifying TensorFlow 1.12 version[3] with CUDA 10.0 and cuDNN 7.4.

DNN models and datasets Our main performance metric is throughput (images/second) of training a DNN model. We use ResNet-152 [16], and VGG-19 [51] with ImageNet [13]. For each DNN model, batch size of 32 is used. For all other hyperparameters, we use the default settings as specified in the benchmark [52] of ResNet-152 and VGG-19.

Resource allocation for virtual workers: Given any heterogeneous GPU cluster, there can be many ways of allocating the resources to the multiple virtual workers. For our experiments, we consider allocation policies within the bounds of our platform. Thus, given the 16 GPUs, HetPipe employs four virtual workers, each of which is configured with four GPUs, along the following three allocation policies.

Node Partition (NP): This policy assigns a node per virtual worker. Thus, each virtual worker is composed of homogeneous GPUs. Consequently, as the nodes are heterogeneous, partitioning of layers for a DNN model is different for each virtual worker. NP results in minimum communication overhead within each virtual worker as communication between GPUs occurs within the same node via PCI-e, rather than across multiple nodes where communication is via InfiniBand. On the other hand, as the performance of each virtual worker varies, a straggler may degrade performance with DP.

Equal Distribution (ED): This policy evenly distributes GPUs from each node to every virtual worker. Thus, every virtual worker is assigned four different GPUs, but every virtual worker has the exact same resources. Thus, model partitioning is the same, and thus, performance will be the same across the virtual workers, which mitigates the straggler problem. However, ED results in high communication overhead within each virtual worker.

Table 3: Resource allocation for the three policies considered

	Node Partition	Equal Distribution	Hybrid Distribution
VW1	VVVV	VRGQ	VVQQ
VW2	RRRR	VRGQ	VVQQ
VW3	GGGG	VRGQ	RRGG
VW4	QQQQ	VRGQ	RRGG

Hybrid Distribution (HD): This policy is a hybrid of NP and ED. For our cluster, a combination of two GPU types are allocated to each virtual worker such that their performances in terms of aggregated computation capability and amount of GPU memory are similar to each other. This choice is made to mitigate the straggler problem while reducing the communication overhead within each virtual worker. As, in terms of computation power, V > R > G > Q and, in terms of the amount of the GPU memory, R > V > Q > G, two virtual workers are allocated VVQQ, while the other two are allocated RRGG, where V, R, G and Q refers to TITAN V, TITAN RTX, GeForce RTX 2060, and Quadro P4000, respectively.

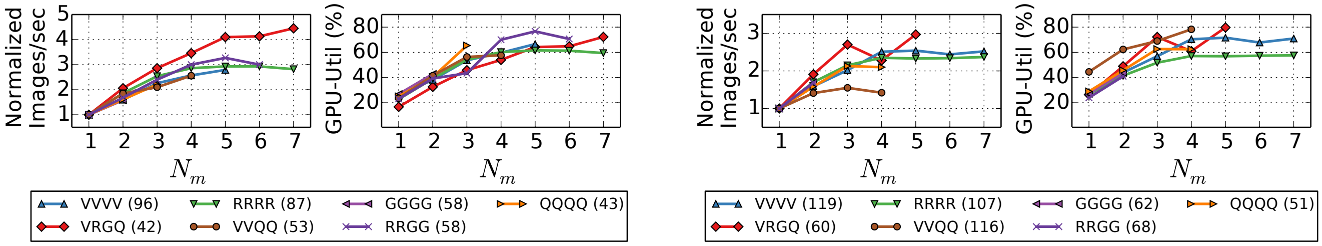
Table 3 shows the resource allocation of each virtual worker for the three resource allocation policies.

Parameter Placement: In our experiments, for DP, we locate the parameter servers, each of which only handles a portion of the model parameters, over all the nodes. For the *default* placement policy, which can be used with all three of our resource allocation policies, we place layers of the model in round-robin fashion over all the parameter servers as in TensorFlow [53]. For ED, however, another policy is possible, which we refer to as 'ED-local'. With 'ED-local', we place the layers of a partition on the parameter server running on the same node, incurring no actual network traffic across the nodes for parameter synchronization. This is possible as the same partition of the model can be assigned *locally* to the GPU on the same node for every virtual worker. For all results reported hereafter, the 'default' policy is used, except for 'ED-local'.

8.2 Performance of a single virtual worker

We first investigate the performance of the 7 different individual virtual workers that are possible according to the allocation schemes in Table 3. Figure 4 shows the throughput over various values of N_m, which is the number of minibatches executed concurrently, in the virtual worker normalized to that of when $N_m = 1$ and the maximum average GPU utilization among the four partitions for ResNet-152 and VGG-19. The numbers shown (in the box) along with the allocation policy are the absolute throughput (images/sec) when $N_m = 1$. Note that some results for larger N_m are not shown. This is because the GPU memory cannot accommodate such situations and hence, cannot be run.

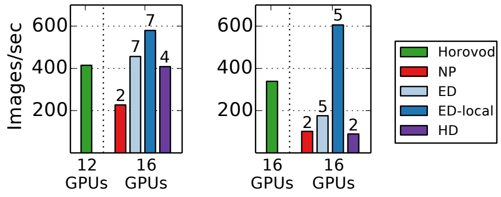
[3] Modified LOC is ~1.5K in the TensorFlow framework and TensorFlow benchmark codes, where most features are added as independent functions.



(a) ResNet-152

(b) VGG-19

Figure 4: Normalized throughput and the maximum average GPU utilization among partitions in a single virtual worker for various resource allocation policies as N_m is varied. The number in parenthesis is absolute throughput (images/sec) when $N_m = 1$ (which is equivalent to the naive MP) for each policy.



(a) ResNet-152

(b) VGG-19

Figure 5: Performance with the three allocation policies when D=0 (The number on bar represents N_m)

From the results, we can see that as N_m increases, normalized throughput of a virtual worker as well as the maximum GPU utilization generally increases. Note that, though not shown, the total GPU memory utilization tends to increase as N_m increases. However, depending on the resource allocation scheme (which results in different partitions of a model) as well as the DNN model, the effect of having larger N_m varies. When a virtual worker is configured with homogeneous GPUs, the average GPU utilization of each partition is similar to each other. However, when it is configured with heterogeneous GPUs, there is a tendency that the GPU utilization of the first or last partition is higher than those of the other partitions. For this configuration, different computation capabilities and memory capacity of the GPUs are considered when partitioning a model. As it is possible that only a small number of layers are assigned to some GPUs, the overall GPU utilization may turn out to be low.

8.3 Performance of multiple virtual workers

Figure 5 shows the throughput of training each model with the three resource allocation policies, where "Horovod" indicates the state-of-the-art DP via Horovod that uses AllReduce communication[4]. In these experiments, for each resource allocation policy, N_m is set such that performance is maximized while every virtual worker uses the same value of N_m as this is the assumption behind HetPipe. For ResNet-152, the whole model is too large to be loaded into a single GPU with G type, and thus, Horovod uses only 12 GPUs.

Table 4: Performance improvement of adding whimpy GPUs (The number in parenthesis presents the total number of concurrent minibatches in HetPipe)

Model	Single GPU [V]	Method	4 GPUs 4[V]	8 GPUs 4[VR]	12 GPUs 4[VRQ]	16 GPUs 4[VRQG]
VGG-19	159	Horovod	164	205	265	339
		HetPipe	300(5)	530(16)	572(20)	606(20)
ResNet-152	112	Horovod	233	353	415	X
		HetPipe	256(5)	516(20)	538(24)	580(28)

The results in Figure 5 show that the performance of DNN training is strongly affected by how heterogeneous GPUs are allocated to virtual workers. From the results, we can make the following observations: First, for VGG-19 whose parameter size is 548MB, the performance of Horovod, which reduces communication overhead for parameter synchronization, is better than those of NP, ED, and HD. However, for ResNet-152 whose parameter size is 230MB, ED and HD, which utilize virtual workers with similar performance, show a bit better or similar performance to Horovod (with 12 GPUs). Second, with NP, training performance of ResNet-152 and VGG-19 is low as N_m is bounded by the virtual worker with the smallest GPU memory. Third, with ED-local, intra-communication occurs between each GPU and the parameter server, significantly reducing communication overhead across the nodes, especially for VGG-19, the model with a large parameter set. For VGG-19, the amount of data transferred across the nodes per minibatch with ED-local (i.e., 103MB) is much smaller than that with Horovod (i.e., 515MB). Thus, the performance of ED-local (which also mitigates the straggler problem) is 1.8× higher than Horovod. For ResNet-152, the amount of data transferred with ED-local (i.e., 298MB) is larger than that with Horovod (i.e., 211MB) because the sizes of output activations to be sent between partitions are large, even though the parameter size is relatively small. However, the throughput of ED-local is still 40% higher than Horovod. This is because Hetpipe allows each virtual worker to process a large number of minibatches concurrently. Compared to NP and HD, ED-local (or ED) usually has larger N_m in each virtual worker, improving throughput.

[4] We use the same minibatch size for all workers of Horovod as the minibatch size is one of the critical factors to the final performance of a trained DNN and adaptive batch sizing will affect convergence [5].

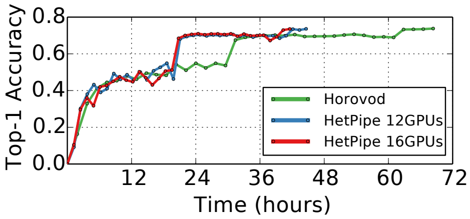


Figure 6: ResNet-152 top-1 accuracy

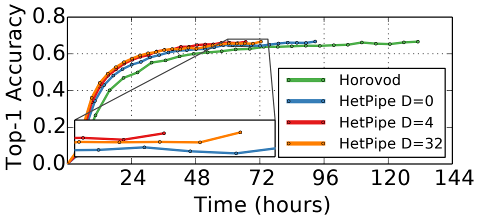


Figure 7: VGG-19 top-1 accuracy

Next, we investigate how the throughput is improved when whimpy GPUs are additionally used for training. Table 4 shows the throughput of VGG-19 and ResNet-152 when DP via Horovod and HetPipe with ED-local are used over different sets of heterogeneous GPUs, and also when a single V GPU is used. For these experiments, HetPipe is configured to use four virtual workers, except for '4 GPUs' where a single virtual worker is used. In the table, the number and type of GPUs used for each experiment are also given. From the results, we can see that the performance of both Horovod and HetPipe increases when additional whimpy GPUs are used for training. With additional GPUs, HetPipe can increase the total number of concurrent minibatches processed, having up to 2.3 times speedup. This scenario can be thought of as an answer to when new, higher end nodes are purchased, but one does not know what to do with existing nodes. The results show that making use of the whimpy systems allows for faster training of larger models.

8.4 Convergence

Our HetPipe based on the WSP model is guaranteed to converge as proven in Section 6. In this section, we analyze the convergence performance of HetPipe with ED-local using ResNet-152 and VGG-19. For our experiments, the desired target accuracy of ResNet-152 and VGG-19 is 74% and 67%, respectively.

Figure 6 shows the top-1 accuracy of ResNet-152 with Horovod (12 GPUs), HetPipe (12 GPUs), and HetPipe (16 GPUs), where D is set to 0 for HetPipe. For the experiments with 12 GPUs, the 4 G type GPUs are not used. When the same set of GPUs are used, convergence with HetPipe is 35% faster than that of Horovod by reducing the straggler problem in a heterogeneous environment and exploiting both PMP and DP. Furthermore, by adding four more whimpy G GPUs, HetPipe improves training performance even more, converging faster than Horovod by 39%.

Figure 7 shows the top-1 accuracy of VGG-19 with Horovod and HetPipe as we vary D to 0, 4, and 32. For the experiments, all 16 GPUs are used. The figure shows that convergence with the BSP-like configuration (i.e., $D = 0$) of HetPipe is roughly 29% faster than that with Horovod. As we increase D to 4, the straggler effect is mitigated and the communication overhead due to parameter synchronization is reduced. Thus, convergence is faster by 28% and 49% compared to $D = 0$ and Horovod, respectively. In this experiment with ED-local (where the training speed of each virtual worker is similar), when D becomes very large (i.e., 32), the throughput remains similar but the convergence performance degrades by 4.7%, compared to $D = 4$. This is because it is unlikely that the clock distance between the fastest and slowest virtual workers becomes as large as 32, but higher global staleness can degrade the convergence performance (similarly discussed in [18]). Note that though not shown, using larger D has a greater effect for HetPipe with NP, ED and HD resource allocation, and the different resource allocations only affect the set of heterogeneous GPUs used for each virtual worker and do not affect the convergence behavior.

We also analyze the synchronization overhead as D is varied. We find that as D increases, the waiting time of a virtual worker to receive the updated global weights decreases. In our experiments, the average waiting time with $D = 4$ is found to be 62% of that with $D = 0$. Furthermore, the actual idle time is only 18% of the waiting time as the virtual worker can continue to proceed in the pipeline while waiting.

9 Discussion

Comparison to PipeDream PipeDream [38], which is the closest related study, optimizes PMP of a single virtual worker, only employing DP for lagging layers within a virtual worker in homogeneous environments. To be adapted to heterogeneous environments, its partitioning algorithm must be extended to consider the different performance and memory sizes of heterogeneous GPUs, various orders of heterogeneous nodes used for a pipeline, and the memory requirement of the GPUs for partitions.

We run the training of ResNet-152 using PipeDream, which is implemented on PyTorch [37], in our heterogeneous GPU cluster described in Section 8.1. Since the partitioning algorithm does not consider heterogeneous GPUs, for each GPU type, we profile ResNet-152, then generate partitions of the model assuming that our cluster is configured with homogeneous GPUs with that type, and finally, measure the throughput of PipeDream with the partitions. All the computed configurations of the pipeline result in a large number of (i.e., 12 or 14) partitions. For example, with Q, the configuration is 4-2-1-1-1-1-1-1-1-1-1-1 indicating that the model is divided into 12 partitions where the first partition is executed by four GPUs with DP, the second one is executed by two GPUs with DP, and so on. For these configurations, we run experiments with various orders of the four different nodes and test using several batch sizes. (Note that we could not run training for some configurations due to out of memory errors.) The best throughput measured using PipeDream is 158. Recall that the throughputs of Horovod (with 12 GPUs) and HetPipe are 415 and 580, respectively. In this case, the performance of PipeDream for ResNet-152 is found to be low as a large number of partitions cause high network overhead, in addition to the sub-optimal partitions. Therefore, with PMP alone (i.e., single virtual worker), the performance benefit may become limited when a model is divided into numerous partitions. Instead of increasing partitions, running DP with multiple virtual workers like HetPipe can improve the parallelism of training and further improve performance in such cases.

Effect of imbalanced partitions Our partitioning algorithm attempts to balance partitions while satisfying the memory requirements. However, depending on the DNN model, computed partitions may be imbalanced. For example, for a model composed of a small number of layers, if one layer takes much longer to execute compared to other layers, the partitions may end up having different execution times. In this case, the performance of the pipeline will be degraded as in any other pipeline-based systems. Note that running DP for the slow partition to have a similar processing rate across all the partitions like PipeDream [38] will be a possible extension of HetPipe.

10 Related Work

Pipelining has been leveraged to improve the performance of machine learning systems [6, 7, 19, 32, 38]. A pipelining scheme is employed to handle expensive backpropagation [7]. Pipe-SGD pipelines the processing of a minibatch to hide communication time in AllReduce based systems [32]. A weight prediction technique is proposed to address the staleness issue in pipelined model parallelism [6]. Detailed comparisons of HetPipe with PipeDream [38] and GPipe [19] are provided in Section 2.2. Note that the feature of overlapping computation and communication, presented in PipeDream [38], will also improve the performance of our system. PipeDream employs the one-forward-one-backward scheduling algorithm for pipeline execution. Sophisticated schedulers that consider various factors such as heterogeneous configurations, the number of partitions, and the number of concurrent minibatches within a virtual worker, can potentially improve the performance of HetPipe. Techniques to optimize learning rates have been studied [15], which can also be applied to HetPipe to help converge faster.

Decentralized training systems that consider heterogeneous environments have also been studied [33, 34]. However, these techniques do not consider integration of DP with PMP, which allows support for large models that do not fit into single GPU memory. In AD-PSGD, once a mini-batch is processed, a worker updates the parameters by averaging them with only one neighbor which is randomly selected [33]. This is done asynchronously, allowing faster workers to continue. In theory, the convergence rate of AD-PSGD is the same as SGD. In principle, the contribution of AD-PSGD is orthogonal with the contributions of HetPipe in that we can extend our HetPipe further by adapting the idea of asynchronous decentralized update in AD-PSGD when there is a bottleneck in the parameter server. When it comes to the experimental evaluations, the performance of AD-PSGD is evaluated for DNN models whose sizes are 1MB, 60MB, and 100MB, which are smaller than the models we consider in HetPipe. For a decentralized training system, Hop [34] considers the bounded staleness and backup workers, and uses CIFAR-10 for performance evaluation on a CNN model.

There have been earlier efforts to employ DP and/or MP for model training. Project Adam uses both DP and MP to train machine learning models on CPUs [8]. Pal et al. combine DP and MP in a similar way as our system, but do not consider pipelining nor heterogeneous GPUs [43]. STRADS leverages MP to address the issues of uneven convergence of parameters and parameter dependencies [27]. FlexFlow considers utilizing parallelism in various dimensions such as sample, operator, attribute and parameters to maximize parallelization performance [23]. Bounded staleness has been explored where Jiang et al. present heterogeneity-aware parameter synchronization algorithms based on the SSP model [24], while Cui et al. analyze the effects of bounded staleness [11].

Hierarchical AllReduce performs the AllReduce operation in two levels [22]. This technique does not solve the straggler problem in a heterogeneous GPU cluster, as master GPUs in the second level will have different GPU types. BlueConnect is an efficient AllReduce communication library considering heterogeneous networks [9]; unfortunately, it also cannot handle stragglers caused by heterogeneous GPUs.

11 Conclusion

In this paper, we presented a DNN training system, HetPipe, that integrates pipelined model parallelism with data parallelism. Leveraging multiple virtual workers, each of which consists of multiple, possibly whimpy, heterogeneous GPUs, HetPipe makes it possible to efficiently train large DNN models. We proved that HetPipe converges and presented results showing the fast convergence of DNN models with HetPipe.

Acknowledgments

We would like to thank our shepherd Saurabh Bagchi and the anonymous reviewers for their invaluable comments. This work was partly supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-14 and Institute for Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-00118 and No.2019-0-00075). Young-ri Choi is the corresponding author.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Amazon. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 2003.

[4] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of COMPSTAT*, 2010.

[5] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 2018.

[6] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. *arXiv preprint arXiv:1809.02839*, 2018.

[7] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined Back-Propagation for Context-Dependent Deep Neural Networks. In *Proceedings of the Annual Conference of the International Speech Communication Association*, 2012.

[8] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System . In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[9] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[10] Ronan Collobert and Jason Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

[11] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[14] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. CatBoost: gradient boosting with categorical features support. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[17] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, 2012.

[18] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2013.

[19] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[20] IBM. CPLEX-Optimizer. https://www.ibm.com/analytics/cplex-optimizer/.

[21] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[22] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shi Shaohuai, and Chu Xiaowen. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[24] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[25] Wenbin Jiang, Geyan Ye, Laurence T Yang, Jian Zhu, Yang Ma, Xia Xie, and Hai Jin. A Novel Stochastic Gradient Descent Algorithm Based on Grouping over Heterogeneous Cluster Systems for Distributed Deep Learning. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2019.

[26] Tian Jin and Seokin Hong. Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[27] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[28] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[30] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2014.

[31] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[32] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[33] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[34] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. Hop: Heterogeneity-aware Decentralized Training. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[35] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zahria, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 2016.

[36] Microsoft. Microsoft Azure Pricing. https://azure.microsoft.com/en-us/pricing/.

[37] msr-fiddle. PipeDream: Generalized Pipeline Parallelism for DNN Training. https://github.com/msr-fiddle/pipedream/.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[39] NVIDIA. GeForce RTX 2060. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060/.

[40] NVIDIA. Quadro P4000. https://www.nvidia.com/en-us/design-visualization/quadro-desktop-gpus/.

[41] NVIDIA. TITAN RTX. https://www.nvidia.com/en-us/titan/titan-rtx/.

[42] NVIDIA. TITAN V. https://www.nvidia.com/en-us/titan/titan-v/.

[43] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *IEEE Micro*, 2019.

[44] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. *arXiv preprint arXiv:2005.14038*, 2020.

[45] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 2009.

[46] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2017.

[47] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

[48] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2011.

[49] Muhammad Saqib, Sultan Daud Khan, Nabin Sharma, and Michael Blumenstein. A Study on Detecting Drones Using Deep Convolutional Neural Networks. In *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2017.

[50] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[51] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[52] TensorFlow. TensorFlow benchmarks. https://github.com/tensorflow/benchmarks/.

[53] TensorFlow. tf.train.replica_device_setter. https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/replica_device_setter/.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[55] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[56] Tian Wang, Yang Chen, Mengyi Zhang, Jie Chen, and Hichem Snoussi. Internal Transfer Learning for Improving Performance in Human Action Recognition for Small Datasets. *IEEE Access*, 2017.

[57] FengLi Yu, Jing Sun, Annan Li, Jun Cheng, Cheng Wan, and Jiang Liu. Image Quality Classification for DR Screening Using Deep Learning. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2017.

[58] Shen-Yi Zhao and Wu-Jun Li. Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee. In *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2016.

[59] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[60] Martin Zinkevich, John Langford, and Alex J Smola. Slow Learners are Fast. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2009.

AutoSys: The Design and Operation of Learning-Augmented Systems

Chieh-Jan Mike Liang[‡] Hui Xue[‡] Mao Yang[‡] Lidong Zhou[‡] Lifei Zhu[*‡] Zhao Lucis Li[⋆‡]
Zibo Wang[⋆‡] Qi Chen[‡] Quanlu Zhang[‡] Chuanjie Liu[○] Wenjun Dai[†]
[‡]*Microsoft Research* [*]*Peking University* [⋆]*USTC* [○]*Microsoft Bing Platform* [†]*Microsoft Bing Ads*

Abstract

Although machine learning (ML) and deep learning (DL) provide new possibilities into optimizing system design and performance, taking advantage of this paradigm shift requires more than implementing existing ML/DL algorithms. This paper reports our years of experience in designing and operating several production learning-augmented systems at Microsoft. AutoSys is a framework that unifies the development process, and it addresses common design considerations including ad-hoc and nondeterministic jobs, learning-induced system failures, and programming extensibility. Furthermore, this paper demonstrates the benefits of adopting AutoSys with measurements from one production system, Web Search. Finally, we share long-term lessons stemmed from unforeseen implications that have surfaced over the years of operating learning-augmented systems.

1 Introduction

Learning-augmented systems represent an emerging paradigm shift in how the industry designs modern systems in production today [33]. They refer to systems whose design methodology or control logic is at the intersection of traditional heuristics and machine learning. Due to the interdisciplinary nature, learning-augmented systems have long been widely considered difficult to build and require a team of engineers and data scientists to operationalize. To this end, this paper reports our years of experience in designing and operating learning-augmented systems in production at Microsoft.

The need of learning-augmented system design stems from the fact that heterogeneous and complex decision-makings run through each stage of the modern system lifecycle. These decisions govern how systems handle workloads to satisfy user requirements under a particular runtime environment. Examples include in-memory cache eviction policy, query plan formulation in databases, routing decisions by networking infrastructure, job scheduling for data processing clusters, document ranking in search engines, and so on.

Most of these decision-makings have been solved with explicit rules or heuristics based on human experience and comprehension. However, while heuristics perform well in general, they can be suboptimal as modern systems evolve. First, since many heuristics were designed at the time when computation and memory resources were relatively constrained, their optimality was often traded for execution cost. Second, since heuristics are typically designed for some presumably general cases, hardware/software changes and workload dynamics can break their intended usage or assumptions. Third, many modern systems have grown in complexity and scale beyond what humans can design heuristics for.

Recent advances in machine learning (ML) and deep learning (DL) have driven a shift in system design paradigm. Various efforts [6, 7, 16, 28, 34, 36, 48] have found success in formulating certain system decision-makings into ML/DL predictive tasks. Conceptually, from past benchmarks, ML/DL techniques can learn factors that impact the system behavior. For example, Cortez et al. [16] reported an 81% accuracy in predicting average VM CPU utilization, which translates to $\sim 6\times$ more opportunities for server oversubscription; Alipourfard et al. [7] reported near-optimal cloud configurations being predicted for running analytical jobs on Amazon EC2.

AutoSys is a framework that unifies the development process of several learning-augmented systems at Microsoft. AutoSys has driven decision-makings with ML/DL techniques, for several critical performance optimization scenarios. These scenarios range from web search engine, advertisement delivery infrastructure, content delivery network, to voice-over-IP client. Not only do these scenarios allow us to gain insights into the learning-augmented design, but they also reveal common design considerations that AutoSys should address.

Contributions. This paper makes the following key contributions, through reporting our years of experience in designing

This work was done when Lifei Zhu, Zhao Lucis Li, and Zibo Wang were interns at Microsoft Research.

and operating learning-augmented systems.

First, Section 2 analyzes the need for adopting the learning-augmented design, with concrete observations from modern systems in production. Due to its architectural similarities to most modern systems, this paper uses web search infrastructure (*Web Search*) as the target system scenario for performance optimization. We characterize sources of system complexity and operation complexity in modern systems, to contribute an understanding of the emergence of learning-augmented system design in industry.

Second, Section 3 describes the AutoSys framework that formulates a system decision-making as an optimization task. AutoSys incorporates proven techniques to address common design considerations in building learning-augmented systems. ***(1)*** To support scenario-specific decision-making, AutoSys employs a hybrid architecture – decentralizing inference plane for system-specific interactions such as actuations and exploration, and centralizing training plane for hosting an array of ML/DL algorithms with generalized abstractions. ***(2)*** To handle ad-hoc and nondeterministic jobs spawned by an optimization task, AutoSys employs a cross-layer solution – prioritizing jobs based on their expected gains towards solving the given optimization task and executing jobs in a container to satisfy heterogeneous job requirements in a resource-sharing environment. ***(3)*** To handle learning-induced system failures due to inference uncertainties, AutoSys incorporates a rule-based engine with hard rules authored by experts to check an inferred actuation's commands and assumptions.

Third, we report long-term lessons stemmed from the years of operations, and these lessons include higher-than-expected learning costs, pitfalls of human-in-the-loop, generality, and so on. Prior to sharing these lessons in Section 5, Section 4 quantifies benefits of the learning-augmented design, on Web Search's key application logic and data stores. Compared to years of expert tuning, Web Search exhibits an 11.5% reduction in CPU utilization for a keyword-based selection engine, 3.4% improvement in relevance score for a ranking engine, 16.8% reduction in key-value lookups for a datastore cluster, and so on. The core of AutoSys is open-sourced on GitHub (*https://github.com/Microsoft/nni*).

2 Background and Motivations

As system performance drives end-user experience and revenue, many modern systems are supported by large teams of engineers and operators. This section shares concrete observations in production, which have motivated the industry to transit to the learning-augmented system design. Particularly, we deep dive into one large-scale cloud system – the web-scale search service, or *Web Search*. Web Search is architecturally representative of modern systems, with fundamental building blocks of networking, application logic, and data stores.

2.1 Overview of Web-Scale Search

This section describes the Web Search design with respect to the fundamental building blocks of modern systems.

Distributed and Pipelined Infrastructure. Web Search realizes a multi-stage pipeline of networked services (c.f. Figure 1), to iteratively refine the list of candidate documents for a user search query. The first stage is *Selection service* which selects relevant documents from massive web indexes as candidates for subsequent Ranking service. It relies on both keyword-based and semantics-based matching strategies, i.e., KSE and SSE. Then, *Ranking service* orders these documents according to their expected relevance to the user query, by running the RE ranking engine. Finally, *Re-ranking service* adds additional web contents that are relevant to the user query, and it re-ranks search results. These additional contents are from sources such as stock and weather, and verticals such as news and images. Suppose the user query contains celebrity names, search results will likely have relevant news and images.

Application Logic. We present three applications implemented with rules, heuristics, and ML-based logic.

First, Selection service's Keyword-based engine (KSE) matches keywords in user queries and web documents, by looking up inverted web indexes. Queries are first classified into pre-specified categories. Each category corresponds to a physical execution plan, or a hand-crafted sequence of sub-plans to specify the document evaluation criteria. For example, one sub-plan can specify whether a query keyword should appear in the web document title/body/URL, and how many documents should be retrieved. Sub-plan knobs determine the trade-offs between search relevance and latency.

Second, Selection service's semantics-based engine (SSE) selects web documents with keywords semantically similar to the user query. The problem can be formulated as Approximate Nearest Neighbor (ANN) search [14, 47] in the vector space where keywords that share similar semantics are located in close proximity. The search strategy is an iterative process, and each step can take on one of the three possible actions: ***(1)*** identifying some anchors in the vector space by looking up the tree, ***(2)*** marking anchors' one-hop neighbors in the neighborhood graph as new anchors, and ***(3)*** terminating and returning the best anchors that we have seen. The action sequence determines how fast SSE returns semantically relevant document candidates.

Third, Ranking service's ranking engine (RE) implements a ranking algorithm based on high-performance LambdaMART [12], which uses Gradient Boosted Decision Trees (GBDT) [20]. GBDT is one of the sophisticated ranking algorithms hosted by Web Search, and each targets different query types, document types, languages, and query intentions. Since GBDT combines a set of sub-models to produce the final results, tuning RE requires data scientists to reason about how tuning each sub-model would impact the overall performance.

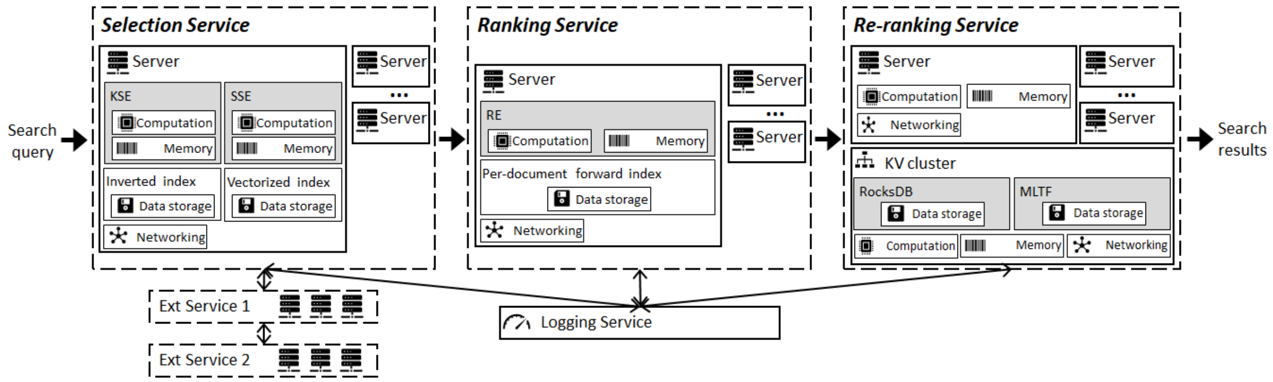


Figure 1: AutoSys drives transitions of several critical engines in Web Search to the learning-augmented design. These engines include *KSE* (Keyword-based Selection Engine), *SSE* (Semantics-based Selection Engine), *RE* (Ranking Engine), *RocksDB* key-value store engine, and *MLTF* (Multi-level Time and Frequency) key-value store engine. Since Web Search is architecturally similar to modern systems in general, AutoSys has also been applied to other production systems at Microsoft.

Data Store. One common data structure of web indexes is the key-value store. Web Search employs both open-sourced RocksDB, and customized solutions such as Multi-level Time and Frequency key-value store (MLTF). MLTF takes key access time and frequency as signals to decide cache evictions.

The index of SSE engine is organized in a mixed structure of space partition tree and neighborhood graph. Space partition tree is used to navigate the search to some coarse-grained subspaces while the neighborhood graph is used to traverse the keywords in these subspaces.

2.2 Sources of System Complexity

Heterogenous Classes of Decisions. Decision-makings in systems can be grouped into three classes: application logic, system algorithms, and system configurations. Each class requires human experts with different skill sets and experience.

First, application logic implements features that fulfill user requirements, so its decision-making process should adapt to user usage. In the case of Web Search, Ranking service hosts hundreds of lightweight and sophisticated ranking algorithms for different user query types and document types. Optimizing these ranking algorithms requires data scientists to have a deep understanding of how different ML/DL capabilities can be combined to match user preferences.

Second, the infrastructure implements system algorithms to better support application requirements with available resources. In the case of Web Search, Selection service has algorithms responsible for compiling user queries into physical execution plans that are specific to underlying hardware capabilities. Optimizing algorithms requires system designers to consider the relationship between application requirements and infrastructure capabilities.

Third, system configurations are knobs for operators to customize systems. Optimizing knobs requires a deep understanding of their combined effects on system behavior [50].

Multi-Dimensional System Evaluation Metrics. Optimizing multiple metrics can be non-trivial if they have different (and potentially conflicting) goals. For instance, Selection service has tens of metrics in different categories: resource usage, response latency, throughput, and search result relevance. Reasoning about the trade-offs among multiple metrics quickly becomes painstaking for humans, as the number of evaluation metrics increases. In some cases, system designers follow a rather conservative rule: improving some metrics without causing other metrics to regress. In fact, any software update in Web Search that can cause search quality degradation should not be deployed, even if it improves some crucial metrics such as the query latency for top queries.

Modern systems can also have meta-metrics that aggregate a set of metrics or measurements over a time period. One example is the "weekly user satisfaction rate" of Ranking service. To optimize these aggregated metrics, system operators need to understand their compositions.

End-to-End and Full-Stack Optimization. Modern systems are constructed with subsystems and components to achieve separation of concerns. Since the end-to-end system performance represents an aggregated contribution of all components, optimizing one component should consider how its outputs would impact others. For example, we have observed that Selection service may increase the number of potentially relevant pages returned, at the risk of increasing spam pages. If the subsequent Ranking service does not consider the possibilities of spam pages, it can hurt user satisfaction.

2.3 Sources of Operation Complexity

Environment Diversity and System Dynamics. While hardware upgrades and infrastructure changes can offer new capabilities, they potentially alter the existing system behavior. An example is how we altered the in-memory caching mechanism design, according to I/O throughput gaps between memory and mass storage medium for different data sizes. Furthermore, hardware upgrades might be rolled out in phases [41], and server resources can be shared with co-located tenants. Therefore, it is possible that instances of a distributed system face different resource budgets.

Modern systems have increasingly adopted tighter and more frequent software update cycles [39]. These software updates range from architecture, implementation, to even data. For example, Selection service has bi-annual major revisions to meet the increasing query volume and Web documents size, or even to adopt new relevance algorithms. And, Re-ranking service can introduce new data structures for new data types, or new caching mechanisms for the storage hierarchy. Finally, software behavior can change with periodic patches, bug fixes [54], and even the monthly index refresh.

Workload Diversity and Dynamics. Interestingly, there can be non-trivial differences among the workloads that individual system components actually observe. The reason is that subsystems can target different execution triggers, or depend on the outputs of others. For example, the list of candidate documents returned by Selection service predominately dictates the workload of Ranking service. And, if a large number of user queries do not have hits in web indexes, Ranking service would have low utilization. The same observation is applicable to the Re-ranking service.

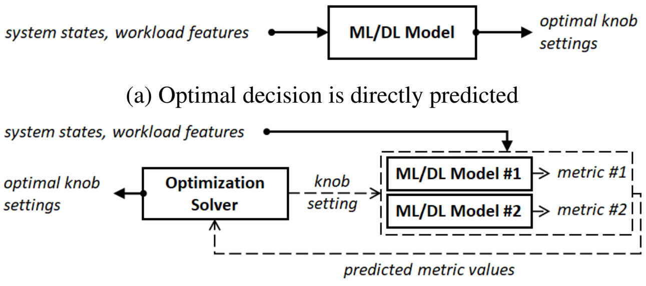
Furthermore, the workload can have temporal dynamics that are predictable and unpredictable, and an example is where the search keyword trend can shift with national holidays and breaking news, respectively.

Non-Trivial System Knobs. Modern systems can expose a large number of controllable knobs to system operators. These knobs include software logic parameters, hardware configurations, actions of an execution sequence, engine selections, and so on. Operation complexity arises from the following observations. A set of knobs can have dependencies [54], i.e., the effect of one knob depends on the setting of another knob. In addition, knobs should be set with the prior knowledge of runtime workloads and system specifications [52]. In the presence of system and workload dynamics, operators need to periodically adjust knob settings for optimal performance. Finally, software parameters can take values of several types: continuous numbers (e.g., 0 - 1,000), discrete numbers (e.g., 1 and 2), and categorical values (e.g., `ON` and `OFF`).

3 AutoSys

We introduce the *AutoSys* framework to unify the development of learning-augmented systems. While AutoSys has driven performance optimization for several production systems at Microsoft, our discussions here focus on Web Search.

Optimization Tasks. In AutoSys, a system decision-making is formulated as an optimization task. In the case of system performance optimization, the output of an optimization task contains optimal values of system knobs. The input consists of system and workload characteristics (e.g., traffic arrival rate). During execution, an optimization task can trigger a sequence of jobs of the following types: ***(1)*** system exploration jobs, ***(2)*** ML/DL model training and inferencing jobs, and ***(3)*** optimization solver. Next, Figure 2 illustrates two use cases of optimization tasks.



(a) Optimal decision is directly predicted

(b) Optimal decision is indirectly predicted

Figure 2: Heterogenous classes of decision-makings can be formulated as ML/DL optimization tasks. This figure illustrates two common realizations of optimization tasks.

Figure 2a illustrates the first case where AutoSys predominately learns from human experts, who handcraft the training dataset containing preferable knob settings for some system states and workloads. In this case, the model takes in system states and workload features as inputs, and directly infers the optimal knob settings. As one example implementation, AutoSys can assign a high reward for these preferable knob settings, and the model can implement value functions to find a policy that maximizes the reward for unforeseen inputs.

Figure 2b illustrates the second case where AutoSys predominately learns from interactive explorations with the target system. By automatically generating system benchmark candidates, AutoSys collects measurements to train models. In this case, the model takes in a knob setting and predicts the expected value of performance metrics. Based on these model predictions, an optimization solver can infer optimal knob settings. An example implementation of model and solver is regression models and gradient descent. For cases where a sequence of step-wise actions is necessary such as Selection service's search query plans, the solver can be based on reinforcement learning.

3.1 Design Principles

AutoSys follows the design principles below, to address common considerations in building learning-augmented systems.

To support scenario-specific decision-makings, AutoSys implements a hybrid architecture. Specifically, a centralized training plane is shared across all target systems, and decen-

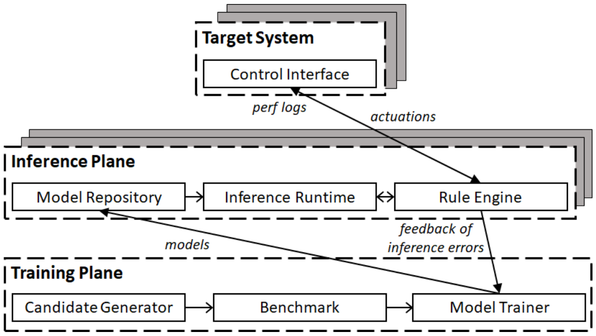


Figure 3: AutoSys framework. It centralizes training plane which hosts an array of ML/DL algorithms, and decentralizes inference plane for system-specific interactions such as actuations and exploration.

tralized inference planes are deployed for each target system. We observe that a centralized training plane promotes sharing data and trained models among scenarios – for example, this can help bootstrapping model training by initializing neural network weights and model hyper-parameters. Decentralized inference planes help distribute inference loads that grow with the system scale, and they also allow scenario-specific customizations such as verification rules.

To manage computation resources, AutoSys implements a cross-layer solution. Specifically, AutoSys abstracts scenarios as optimization tasks, and allows target systems to prioritize jobs spawned by their tasks. Unifying learning-augmented scenarios allows computation resources to be flexibly shared, especially since tasks are ad-hoc and non-deterministic. First, tasks are triggered in response to system dynamics, which might not exhibit a regular pattern. Second, jobs are determined at runtime according to the optimization task progress.

To handle learning-induced system failures, AutoSys implements a rule-based engine to validate actuations. Since most models mathematically encode knowledge learned, existing verification tools might not be applicable. On the other hand, rules are human-readable and human-verifiable.

3.2 Framework Overview

AutoSys executes an optimization task by spawning a number of jobs: ***(1)*** system exploration jobs, ***(2)*** ML/DL model training and inferencing jobs, and ***(3)*** optimization solver. Figure 3 shows the overall AutoSys framework to support these jobs.

Training Plane. The training plane implements features to support both system exploration jobs and ML/DL training jobs. Figure 4 shows the training plane workflow. The first step is candidate generation, which generates knob values to benchmark for the purpose of building up the training dataset. Considering the costs of running system benchmarks, the key is to balance the number of candidates and the model accuracy. Generation algorithms are wrapped in Tuner instances, and we

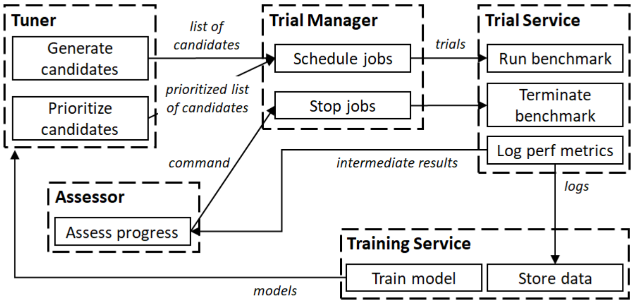


Figure 4: Workflow of training plane. System exploration jobs are wrapped in a Trial object, which collects system benchmark outputs for training models in the Training Service.

have implemented algorithms based on TPE [9], SMAC [26], Hyperband [30], Metis [31], and random search.

The second step is to benchmark configuration candidates. Trial Manager abstracts each system benchmark as a Trial object – the Trial object has fields holding ***(1)*** knob configurations, ***(2)*** execution meta-data: the command to run binaries and even ML/DL models (e.g., RE's hyper-parameter tuning), and metrics to log, ***(3)*** resource requirements (e.g., the number of GPU cores). In the case of KSE, SSE, MLTF, RocksDB engines, their Trial instances point to both the system executable and workload replay tool. The replay tool feeds a pre-recorded workload trace to the executable. In contrast, RE engine has a different goal of optimizing a ranking model's hyperparameters, its Trial instance contains the model and the dataset location. And, invoking `updateConfigs` updates model hyperparameters.

Table 2 presents the Trial Manager API. Invoking `startTrial` submits a Trial instance to Trial Service. At any time, `updateConfigs` can be called to change knob settings, and `getMetrics` can be called to retrieve metric measurements. A Trial can optionally assess its intermediate benchmark results, to decide whether it should terminate the benchmark early. To do so, Trial sends intermediate results to an Assessor instance implementing early-stopping criteria. If the criteria is met, Assessor can invoke Trial Manager's *stopJob*.

The third step is model training. Upon the completion of Trial instances, `getMetrics` outputs are merged with the corresponding knob settings to form a tabular dataset, for Training Service to train models. Historical results are optionally stored in data store, for model re-training or data sharing among similar scenarios.

Inference Plane. The inference plane implements features to support ML/DL inferencing jobs and optimization solver. Taking the current system states and workload characteristics as inputs, these jobs infer optimal actuations. These jobs are typically triggered by events, which are predefined by system operators to support service level agreements. For example, if the target system's workload changes (e.g., an increase in search queries per second), performance drops (e.g., a drop

Name	Description
`tuner.updateSearchSpace(args)`	Specify search space. *args* is a list of system knobs' names, value types, and value ranges.
`candidates = tuner.generateCandidates()`	Generate and return a list of configuration candidates.
`tuner.generateModel()`	Train the Tuner instance's model.

Table 1: Tuner instance API

Name	Description
`trial = submitTrial(args)`	Submit and deploy a benchmark trial. *args* include a configuration candidate, execution meta-data, and scheduling meta-data.
`trial.startTrial()`	Start a benchmark trial.
`trial.stopJob()`	Stop a benchmark trial.
`trial.updateConfigs(args)`	Update a trial's configuration candidate. *args* is a configuration candidate.
`measurements = trial.getMetrics()`	Return perf measurements of a trial.

Table 2: Trial Manager API

in tail latencies), or models update, inference plane initiates optimization tasks to make system tuning decisions. Furthermore, inference plane can be configured to directly actuate the target system, or simply inform system operators as suggestions. Finally, the inference plane can relay online system performance measurements to the training plane, for training.

3.3 Ad-hoc and Nondeterministic Jobs

In contrast to traditional systems, learning-augmented systems introduce jobs that are difficult for system operators to provision beforehand. First, optimization tasks are *ad-hoc* – they are triggered in response to adapting to system and workload dynamics, which might not exhibit a regular pattern. Second, optimization tasks have *nondeterministic* requirements – they spawn system exploration jobs and ML training jobs according to the optimization progress at runtime. To this end, AutoSys implements mechanisms to prioritize, schedule, and execute these jobs.

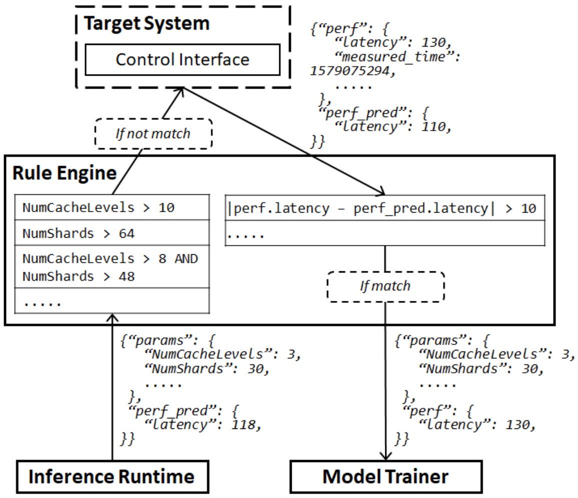
Job Prioritization. The tuner can prioritize the list of candidates in `generateCandidates`, to highlight benchmarks that are expected to subsequently improve model accuracy the most. Unnecessary benchmarks waste time and resources, especially for systems that require warm-up (e.g., in-memory cache warm-up). In this mode, training plane iterates between candidate generation, candidate prioritization, and model training.

We illustrate some of the candidate prioritization strategies that AutoSys Tuners have implemented. First, since the Metis Tuner uses Gaussian process (GP) regression model, it leverages GP's capability to estimate the confidence interval of its inference. A larger confidence interval represents lower inference confidence. And, it prioritizes candidates by sorting their confidence interval in descending order. Second, the TPE Tuner maintains two mixture models to learn the distribution of top-performing knob combinations [9]. It computes how likely a candidate belongs to this distribution, and prioritizes by sorting likelihood scores in descending order.

Job Scheduling. Trial Manager schedules Trials according to priorities and available resources, and passes this information to underlying infrastructure [3, 49]. Trials can impose heterogeneous resource requirements to support their corresponding decision-making scenarios. Taking system exploration jobs as an example – benchmarking ML/DL-learned system components in RE benefits from access to ML/DL acceleration hardware such as GPUs, but benchmarking MLTF KV engine must take place with SSDs. We note that scheduling learning jobs also have similar considerations. For learning approaches based on neural networks, their training jobs can be scheduled to machines with GPUs and TPUs [1] for acceleration. Some learning approaches such as Metis maintain a collection of ML models, and their training and inference time can significantly benefit from multiple CPU cores.

Job Execution. In addition to natively running system exploration jobs (i.e., Trial instances) on real machines, AutoSys also supports containers such as Docker [2]. The container image packages a Trial's software dependencies including the target system's binaries and libraries. Containers benefit AutoSys in the following ways. First, containers can be started and stopped to share hardware resources among multiple target systems. Second, previous efforts reported that containers exhibit a much lower overhead than virtual machines [19]. This improves the benchmark fidelity, hence AutoSys's training data quality. Second, the capability of deploying an image on heterogeneous machines easily enables benchmarking a target system under different hardware environments.

In addition to allocating short-lived containers that run only one benchmark, AutoSys also offers long-lived containers. For Trials with multiple benchmarks, long-lived containers effectively amortize the cost of initializing and loading the image. Furthermore, if consecutive benchmarking jobs need to share states (e.g., warmed-up caches) and data (e.g., weight sharing for tuning ML/DL model hyperparameters), contents in memory can be retained and reused.



```
{"perf": {
    "latency": 130,
    "measured_time":
1579075294,
    .....
 },
 "perf_pred": {
    "latency": 110,
}}
```

```
{"params": {
    "NumCacheLevels": 3,
    "NumShards": 30,
    .....
 },
 "perf_pred": {
    "latency": 118,
}}
```

```
{"params": {
    "NumCacheLevels": 3,
    "NumShards": 30,
    .....
 },
 "perf": {
    "latency": 130,
}}
```

Figure 5: Workflow of rule engine in inference plane.

3.4 Learning-Induced System Failures

Being stochastic in nature, ML/DL inference exhibits some degrees of uncertainty, and this uncertainty can lead to learning-induced system failures or suboptimality. While failures are not unique to learning-augmented systems [8], handling them requires a different approach for the following reasons. First, ML/DL models mathematically encode knowledge learned from the training data, the meaning of their internal weights is not interpretable to humans and common formal verification techniques. Second, as models autonomously learn from the training data, it is difficult to assess whether a dataset would guarantee a model to fully learn a particular concept.

Since it is hard to formally verify ML/DL correctness, AutoSys opts to validate ML/DL outputs with a rule-based engine. These validation rules are authored by operators, and the rule engine functions as a blacklist. Each rule specifies conditions of a violation to catch, and it has the following format: $(Predicate_1 \ AND|OR \ Predicate_2 ...)$. A predicate is one variable value comparison with operators such as $==$, $!=$, $>=$, $>$, $<=$, and $<$. Conceptually, AutoSys maintains the following two rulesets for each target system.

First, ruleset validates ML/DL actuations, or inference runtime outputs as illustrated in Figure 5. In addition to validating parameter value constraints, if certain system states have been known to cause failures (from either past experience or bug reports), system operators can prevent those configurations from being applied. Rules can also encode knob dependencies – an example is the multi-tenant setup where the total memory allocated to all tenants must not exceed a budget, and the blacklist rule can be written as $capacity_1 + capacity_2 > 1024$.

Second, ruleset checks the actuation feedback, or target system outputs as illustrated in Figure 5. One use case in our deployments is to check discrepancies between actual system states and ML/DL inference. Specifically, if the predicted performance (of an actuation) significantly differs from the actual performance measurement, this feedback is relayed to the training plane as additional training data. An example rule is $|perf.latency - perf_p red.latency| > 10$ in Figure 5.

3.5 Extensibility

Since tuning scenarios can vary in requirements, we design AutoSys to be extensible through the Tuner abstraction. Tuner is agnostic to specific candidate generation algorithms, and it provides APIs to wrap the underlying ML/DL details (c.f. Table 1). After a Tuner instance is instantiated, users can specify its search space by invoking `updateSearchSpace` method. The method argument is a list of system knobs' names, value types, and value ranges. Invoking `generateCandidates` generates a list of configuration candidates to be benchmarked for model training. Then, after benchmarks complete, AutoSys invokes `generateModel` to train and update the Tuner instance's model in Training Service.

We have implemented Tuners based on algorithms including TPE [9], SMAC [26], Hyperband [30], Metis [31], and random search. Our implementation of `updateSearchSpace` allows system operators to specify each parameter's expected value type: `choice` (e.g., categorical values for KSE engine's `RankingStreams` parameter), `uniform` (e.g., continuous number within a range for RE engine's `LearningRate`), `randint` (e.g., integers between within a range for RocksDB engine's `WriteBufferSize` parameter), and so on. Finally, for model-less algorithms such as random search, it is not necessary to implement `generateModel`.

3.6 Implementation

Our current implementation comprises ~18,205 lines of Python code (Tuner: 5,427, Assessor: 1,392, Trial Manager: 35, Trial Service: 28), ~12,852 lines of TypeScript code (Trial Manager: 3,283, Trial Service: 6,638), and ~13,344 lines of code in other languages. We have implemented Tuners for an array of popular optimization algorithms such as TPE (Tree-structured Parzen Estimator) [9], SMAC (Sequential Model-based Algorithm Configuration) [26], Hyperband [30], Metis [31], anneal, naïve evolution, grid search, and random search. We have also implemented two early-stopping algorithms based on median stop [23] and curve fitting [18]. Our current implementation supports the following Trial Service realizations: local machine, remote servers, several Kubernetes-based platforms, and several internal experimental platforms. We have open-sourced the core of AutoSys on GitHub (*https://github.com/Microsoft/nni*).

4 Production Deployment Measurements

This section presents production measurements of Web Search, and Table 3 summarizes key results. The goal is to

	Search space size	**Tuning time**	**Key results (vs. long-term expert tuning)**
Keyword-based Selection Engine (KSE)	$O(1000^n)$	1 week	Up to 33.5% and 11.5% reduction in 99-percentile latency and CPU utilization, respectively
Semantics-based Selection Engine (SSE)	Action sequences	1 week	Up to 20.0% reduction in average latency
Ranking Engine (RE)	$O(10^n)$	1 week	3.4% improvement in NDCG@5
RocksDB key-value cluster (RocksDB)	$O(100^n)$	2 days	Lookup latency on-par with years of expert tuning
Multi-level Time and Frequency key-value cluster (MLTF)	$O(100^n)$	1 week	16.8% reduction on avg in 99-percentile latency

Table 3: Summary of adopting learning-augmented design to tune various systems of Web Search (c.f. Section 2.1). We compare key results to the previous practice of manual tuning by human experts over the years. n represents the number of parameters.

quantify benefits of adopting learning-augmented system design in terms of ***(1)*** tuning effort reduction and ***(2)*** system performance improvement.

4.1 Tuning Application Logic

This subsection considers both cases of tuning application logic in a single step and in a sequence of step-wise actions. Specifically, we present measurements from Selection service's KSE engine and SSE engine.

Performance Gain for KSE Engine. KSE engine exposes the following key knobs. Each execution plan consists of a hand-crafted sequence of sub-plans. Each sub-plan has a categorical parameter, *RankingStreams* (`title`, `body`, `anchor`, and `URL`), that specifies document fields that a query keyword should appear in. In addition, it has an integer parameter, *MaxSeekCount* (1 - 1000), that dictates the maximum number of documents the sub-plan should examine. These parameters determine the trade-off between Selection service effectiveness and latency – while a large *MaxSeekCount* potentially increases the number of document candidates for ranking, it also increases the Selection service latency. Depending on the number of execution plans, there can be up to 20 controllable knobs and parameters.

Metrics of interests include per-query latency, CPU utilization, and relevance. As Selection outputs an un-ranked list, we use the popular NCG (Normalized Cumulative Gain) score to quantify the overall relevance, and this is a variant of NDCG [5] that does not consider position-based discounting. Importantly, the higher the NCG, the more likely users will click the corresponding search result. For KSE, the optimization target is to reduce latency and CPU utilization, while keeping relevance score the same.

We optimized KSE for the image and video domain. Web Search divides the image and video domain into several segments: generic, tail (e.g., lower popularity), regions (e.g., US market), and so on. With production workloads, we ran the TPE (Tree-structured Parzen Estimator) Tuner for a week, on a machine with 2.1 GHz CPU (with 8 cores) and 16 GB RAM. Compared to years of expert-tuning, we highlight the following improvements. For the image domain, AutoSys lowered KSE 99-percentile latency by another 16.9% - 33.5%, and CPU utilization by another 9.0% - 11.0%. For the video domain, compared to expert-tuned configurations, AutoSys lowered KSE 99-percentile latency by another 19.4% - 29.7%, and CPU utilization by another 10.1% - 11.5%. These improvements represent a Selection latency reduction up to 33 msec; for reference, many companies have reported an ~1.0% revenue gain from reducing the end-to-end search engine latency by 100 msec.

Performance Gain for SSE Engine. SSE engine optimization concerns with deciding the action for each step of the execution sequence. In contrast to other engines in Web Search, SSE requires a correct ordering of actions. The problem can be formulated as the Approximate Nearest Neighbor (ANN) search [14, 47] in the vector space. As we mentioned before, given a user query, each step of SSE chooses one of the three possible actions: ***(1)*** identifying some anchors in the vector space by looking up the tree, ***(2)*** making anchors' one-hop neighbors in the neighborhood graph as new anchors, and ***(3)*** terminating and returning the best anchors that we have seen. At each step, SSE provides the following system states and environment features for the decision-making: the number of distance calculations between candidates and the query so far, the number of tree searches so far, whether top K candidates have been updated in the last T actions, and the average distance between current top K candidates and the query.

We implemented a reinforcement learning Tuner with tabular based models (Q-tables). Reinforcement learning excels in discovering the action sequence that would maximize the overall reward. We define the reward of step t as a trade-off between relevance gain and latency cost: $R_t = \alpha \times relevance\_gain_t - \beta \times latency\_cost_t$ (α and β are hyperparameters). Under Web Search production workload, we observed that learned execution sequences are able to achieve an average of 20.0% reduction in latency while keeping the relevance score the same.

Additional Consideration: Actuation Granularity. While setting up AutoSys for KSE, we encountered the question of actuation granularity – *should AutoSys generate coarse-grained actuations (i.e., one actuation for all system instances) or fine-grained actuations (i.e., one actuation for each system instance, user segment, region, and so on)*? Coarse-grained actuations impose less computation loads, but fine-grained actuations potentially offer higher performance gains. Unfortunately, the real-world value of learning-augmented design diminishes with either high learning cost

	RS	MS_1	MS_2	MS_3	MS_4
Image-generic	`U,T,B`	300	6	9,00	160
Image-tail	`U,T,B`	484	10	6,30	130
Image-US	`U,T,B,C`	245	50	4,80	90
Video-generic	`U,T,C`	220	160	6,50	10
Video-tail	`U,T,C`	125	30	6,40	180

Table 4: Optimal decisions should vary with workload diversity. This table illustrates the optimal configuration of key KSE knobs for several segments of the image and video search domain. *RS* and *MS* are the abbreviated name for *RankingStreams* and *MaxSeekCount* parameters, respectively.

or low performance gain.

To balance the trade-off, we experimented with three levels of granularity: system-wide, per-instance, and per-search-segment. Due to restrictions imposed by the production environment, we performed exploration jobs on a random selection of 3 instances in each search segment. Results suggested that per-search-segment granularity best balances the trade-off for KSE. Table 4 illustrates the best-performing knob configurations for five popular segments, and there is a noticeable variance in their knob settings.

4.2 Tuning ML Algorithms

This subsection considers tuning system components that host ML/DL algorithms. Specifically, we present measurement from Ranking service's RE engine.

Performance Gain for RE Engine. RE engine runs a set of decision trees, or random forest. Its key controllable knobs include *LearningRate*, *NumberOfLeaves*, *MinimumDocsPerLeaf*, *NumberOfTrees*, and so on. *LearningRate* takes a continuous number (0.01 - 0.99), for adjusting gradient descent speed that trades off between learning convergence time and accuracy. *NumberOfLeaves* takes an integer (10 - 5,000), for adjusting the maximum number of base tree leaves, which relates to the model's learning capability. *MinimumDocsPerLeaf* takes an integer (5 - 1,000), for adjusting the minimum number of documents in a leaf. *NumberOfTrees* takes an integer (5 - 100), for adjusting the number of decision trees. In total, there are approximately 5×10^8 possible parameter combinations in the configuration space.

The optimization metric is NDCG (c.f. Section 4.1), and we ran the TPE (Tree-structured Parzen Estimator) Tuner in AutoSys for one week, on a machine with 2.1 GHz CPU (with 8 cores) and 16 GB RAM. Compared to years of expert-tuning, we highlight the following improvements (evaluated on a production workload containing 150K queries and 2.5M URLs): AutoSys improved NDCG@1 (i.e., top 1 result's NDCG score) by another 2.9%, NDCG@2 (i.e., top 2 results' NDCG score) by another 3.4%, NDCG@3 by another 3.4%, NDCG@4 by another 3.4%, and NDCG@5 by another 3.4%. We note that ranking relevance has a direct correlation with conversion rate (e.g., ads clicking).

Additional Consideration: Human-in-the-Loop. We note that solving the combinatorial optimization from a total of 5×10^8 possibilities is theoretically doable, but it might not be practically feasible. With RE, we took advantage of human knowledge of the engine design, and reduced the value range of several parameters during the process. Interestingly, we have encountered cases where information from humans unintentionally misled learning or caused unexpected consequences, and Section 5 shares these cases.

4.3 Tuning Data Store

Through both RocksDB engine and MLTF engine in Reranking service, we demonstrate data store optimization.

Performance Gain for RocksDB Engine. From years of operation, Web Search operators selected the following key knobs to optimize RocksDB read and write throughputs (in MB per second): *WriteBufferSize* (1 - 96 MB), *BlockSize* (128 - 2,000 KB), *Level0FileNumCompactionTrigger* (2 - 64), and *MaxBackgroundJobs* (1 - 45). Details of these knobs are available online [4]. We used the Metis Tuner because gaussian process models have been shown to be effective for tuning databases [6].

We allocated a two-day computation budget (on an 8-core 2.1 GHz CPU), for AutoSys to search for the optimal configuration with respect to a 5-day trace of production Web Search traffic. AutoSys improves the maximum write throughput to 50.36 MB per second, which matches the throughput achieved by Web Search operators' years of manual tuning. This result demonstrates that AutoSys can significantly reduce the amount of human efforts.

Performance Gain for MLTF Engine. MLTF (Multi-level Time and Frequency) KV engine has the following key knobs. There are *NumCacheLevels* (1 - 10) cache levels. A cached object can move up a level if it has been queried *CachePromotionThreshold* (1 - 1,000) times. Top *NumInevictableLevels* (0 - 9) cache levels can be specified as being inevictable. Furthermore, MLTF does not immediately admit large keys with an object size larger than *AdmissionThreshold* (1 - 1,000) bytes, but it first holds them in a shadow buffer of *ShadowCapacity* (1 - 10) MB. Then, keys in the shadow buffer are moved to the cache only if they have been queried more than *ShadowPromotionFreq* (1 - 1,000) times. The dataset partition on each server is divided into *NumShards* (1 - 64) shards. The metric of interests is the 99-percentile query latency. Due to the noise in latency measurements, we used the Metis Tuner. We collected measurements from one production cluster whose servers have a 512 MB in-memory cache and SSD.

With measurements from a 14-day window, we try to answer the question, *can AutoSys continuously maintain optimal system performance over time?* AutoSys generated a new configuration every two hours to adapt to workload dynamics.

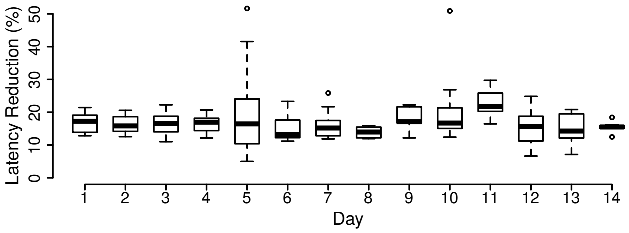


Figure 6: The figure summarizes the 99-percentile latency reduction over a 14-day window, as compared to the static default configuration. AutoSys periodically tuned MLTF every two hours, so we have 12 actuation feedbacks per day.

Figure 6 summarizes the latency reduction for each day, as compared to the default configuration from human operators. We highlight the following observations. First, AutoSys lowered the 99-percentile latency by an average of 16.8%. Second, if the workload changes too frequently, AutoSys might not be able to always update the system configure in time.

Additional Consideration: System Measurement Quality. Noise and outliers are the common factors that system engineers typically consider in terms of measurement quality. In fact, as mentioned above, they are the reason that we used the Metis Tuner for MLTF. Interestingly, while setting up AutoSys for RocksDB, we encountered another factor of system measurement quality – imbalanced measurements. In imbalanced datasets, data points are not roughly equally distributed among all classes of behavior (e.g., read vs. write requests). For example, in one RocksDB scenario, the workload trace is significantly skewed, and writes significantly dominate reads. As a result of using this trace for training, AutoSys optimization tasks frequently produced actuations that sacrificed read throughput for write throughput. These actuations might not be acceptable in the real world.

5 Long-Term Lessons Learned

Although AutoSys addresses common design considerations (c.f. Section 3), unforeseen implications have surfaced over years of operation. They reveal roadblocks in the transition to learning-augmented system design, from the perspective of system operators.

5.1 Higher-Than-Expected Learning Costs

While we anticipated model training would incur some costs, these costs sometimes exceed our expectations due to how operators set up models. The common approach is to model an entire system deployment as one black box. Since ML/DL models directly learn from the observed system execution behavior, operators are freed from worrying about non-trivial component interactions and resource contentions within the deployment. This benefit of simplicity is attractive because interactions and contentions are unavoidable in modern systems – if multiple service instances are deployed on the same server, they would contend for computation and I/O resources, especially on over-subscribed servers. Even for single-instance servers, instances share network resources, job dispatcher, etc.

The first unapparent trade-off of modeling an entire system deployment as one black box is *re-training cost*. Compared to traditional systems, most modern systems are designed to be elastic. Individual instances can be created and destroyed on demand, and they can run on heterogeneous hardware as required. Unfortunately, any changes to the deployment setup would invalidate model assumptions and cause the trained model to be irrelevant. In our example above, system setup changes include the number of co-located instances on a server and instance migration. Re-training models for modern systems can be costly. Complex systems require complex ML/DL models, which tend to be difficult to train and require a large amount of training data.

The second unapparent trade-off of modeling an entire system deployment as one black box is *exploration cost*. Considering optimizing the job completion time for a cluster of hundreds (or even thousands) workers and job dispatchers, if we consider individual nodes' CPU utilization, the model would already have hundreds of inputs to learn. Furthermore, preparing training datasets for this model scale can be challenging: ***(1)*** testbeds rarely match the target system's hyper scale in the real world, and ***(2)*** exploratory actuations on critical systems in production are prohibitive.

To mitigate higher-than-expected learning costs, one ongoing effort is to take advantage of the target system's software modularity [40]. Software modularity emphasizes separating code functionality to promote maintainability. Similarly, instead of modeling an entire system deployment with a monolithic model, we modularize the learning task into composable units of learning assignments. One realization is to dedicate a model to learn a subsystem or a component. Considering a content-aggregation application that queries two local key-value stores for images and videos in series, we can have separate latency-predicting models for these stores, M_{image} and M_{video}. And, the end-to-end latency can be computed by aggregating outputs of M_{image} and M_{video}. Due to the separation, each model has less to learn and can be re-trained independently. Furthermore, if the application is updated to aggregate new content types, additional models (e.g., M_{text}) can be added without updating M_{image} and M_{video}.

We acknowledge that software design modularity might not always be the appropriate level of modularity, especially that software modularity is typically based on the criteria of code functionality and maintainability, rather than learning complexity. This process is currently a manual trial-and-error process for individual systems, and we are accumulating experience to standardize the methodology.

5.2 Pitfalls of Human-in-the-Loop

Senior engineers and operators likely have a wealth of knowledge and experience on the target system, which can guide AutoSys optimization tasks. This subsection describes cases where information from humans unintentionally misled learning or caused unexpected consequences.

First, human experts can inject biases into training datasets, by providing a large number of labeled data points for certain search space regions. This is possible if human experts are already familiar with these regions. As a result, models would exhibit an uneven distribution of uncertainties. For optimization algorithms that tend to exploit regions with lower uncertainties, e.g., Expected Improvement (EI) [44], decisions would likely lean towards regions labeled by human experts. While the academic community has investigated data bias in the context of classification (e.g., images [46]), learning-augmented systems also rely on regression. Our current practice is to advise operators to mix human-labeled datasets with random exploration.

Second, human experts can write conflicting specifications for optimization tasks. Specifically, human experts can help AutoSys narrow down the search space by specifying the *valid* value ranges of each configuration knob, and an example is RE described in Section 4.2. At the same time, they can specify *invalid* configurations for the rule engine to check optimization task outputs. Due to human errors, if the invalid space completely covers the valid space, any outputs would effectively be rejected by AutoSys. Our current practice is to run a tool to check this overlapping condition.

5.3 Closed-Loop System Control Interfaces

We have worked with many production systems that lack closed-loop control interfaces. The closed loop refers to how AutoSys actuates a system to achieve optimality, based on the current system feedback. To this end, not only do modern systems need interfaces to accept external actuations, but they should also have well-defined interfaces that abstract system measurements and logs in a way of facilitating learning.

We describe common issues that motivate this need. First, some systems distribute configurable parameters and error messages over a set of not-well documented configuration files and logs [42]. And, directly modifying configuration files means that the system can not enforce value checks or provide immediate feedbacks. Second, parsing raw logs can be time-consuming, especially if system components disagree on a unified logging format or excessively log [27]. Third, many system feedbacks are not natively learnable, e.g., stack traces and core dumps.

To this end, we have been customizing closed-loop control interfaces for individual systems. Our current practice consists of the following steps. We ensure interfaces contain accessors for all configurable knobs and also accessors for system metrics. The latter output system measurements in the format of time-series values, which capture system measurements since the last AutoSys actuation. Furthermore, control interfaces implement mechanisms to remove system-specific data outliers (e.g., Gaussian noise and spikes), to improve the quality of system benchmark measurements as training data.

5.4 Applicability to Other Systems

This subsection summarizes our experience in applying AutoSys to systems other than Web Search. AutoSys works extremely well in a well-controlled learning environment where high-quality workload traces can be easily collected from the target system, and training can take place offline on high-fidelity testbeds or simulators. Interestingly, many critical scenarios already have the infrastructure to satisfy these strict requirements for debugging purposes.

Many target systems have a more relaxed learning environment. First, real-time exploration can be slow, especially for systems that require warm-up (e.g., in-memory cache). For Tuners based on Bayesian optimization or reinforcement learning, training can take a long time. Our current practice is to run multiple Trials for multiple concurrent benchmarks, at each iteration of exploration. Second, online in-situ exploration with production systems can be restricting and even prohibitive. our current practice is to construct base models offline by running exploration on testbeds or simulators, and then fine-tune models online with live traffic. This practice is useful, especially for systems where individual instances exhibit different workload characteristics. Finally, Section 5.1 discusses cases of frequent model retraining, due to various types of dynamics.

6 Related Work

There are efforts on exploring and demonstrating the potential of learning in solving certain system challenges. Building on these efforts, AutoSys takes a step towards unifying the development of learning-augmented systems. Anticipating growing system scale and complexity, Self-* [22] stated a vision of autonomic computing that satisfies a collection of "self-*" properties, and proposed a conceptual model. Recent efforts include learning index structures and memory access patterns [25, 28], optimizing data query evaluations [37], system performance tuning [7, 31], database configuration tuning [6, 13], placing deep learning computational graphs onto hardware device [35, 36], anomaly detection [21, 29, 55], etc.

There are efforts on building general-purpose predictive service. Resource Central [16] is a predictive service to drive Azure's VM scheduler, and it builds random forest and XGBoost models from past VM telemetry, rather than interactive explorations. Vizier [23] is a general-purpose black-box optimization service, and it has enabled tasks such as parameter tuning at Google. Vizier implements Bayesian optimization to learn the search space through interactive explorations. Clipper [17] is a general-purpose low-latency prediction serving system which introduces a modular architecture to simplify model deployment across frameworks. However, these efforts do not consider some of the challenges in operationalizing learning-augmented systems such as interactive explorations, learning-induced system failures, and so on.

Some AutoSys components are inspired by decades of research and experience in the system community. Many efforts heavily focus on system challenges to support learning tasks [15, 38, 49], and Berkeley shared their views of system challenges for artificial intelligence (AI) [45]. Related to control interfaces, interfaces and methods for controlling and exploring systems state are used for implementation-level model checking (e.g., MaceMC [24] and Modist [51]). One approach to drive automating system performance tuning is interactive exploration. Fuzz testing has been effectively used in generating inputs to induce unexpected software behavior [10, 32, 53], and there is a rich literature on software testing and system debugging. Inspired by the idea of composable AI [45], we are exploring how assembling previously trained models can scalably model large-scale systems.

Finally, some AutoSys components are inspired by research in the ML/DL community. Examples include online learning [11], continual learning [43], and so on.

7 Conclusion

This paper reports our years of experience in designing and operating learning-augmented systems at Microsoft. To unify the development process of these systems, we introduce the AutoSys framework that addresses common design considerations. Furthermore, we present production measurements and discuss long-term lessons learned from operating one such system, Web Search. Going forward, we will study how learning-augmented systems should evolve models over time, and how end-to-end and full-stack system optimization can be safely carried out in practice.

Acknowledgments

We thank anonymous reviewers and our shepherd, Prof. Ana Klimovic, for their extensive comments and suggestions. We also thank our colleagues at Microsoft, for their help in deploying AutoSys: Mingqin Li (Bing Platform), Wei Li (Bing Platform), Haidong Wang (Bing Platform), Yuanchi Yan (Bing Platform), Chao Zhang (Bing), Qiang Zhang (Bing), and Wenjin Zhang (Bing).

References

[1] Cloud TPU. http://cloud.google.com/tpu/.

[2] Docker. http://www.docker.com.

[3] Resource Scheduling and Cluster Management for AI. http://github.com/microsoft/pai.

[4] RocksDB Tuning Guide. http://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[5] Agichtein, E., Brill, E., and Dumais, S. Improving Web Search Ranking by Incorporating User Behavior Information. In *SIGIR* (2016), ACM.

[6] Aken, D. V., Pavlo, A., Gordon, G. J., and Zhang, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD* (2017), ACM.

[7] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (2017), USENIX.

[8] Benson, T., Akella, A., and Shaikh, A. Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services. In *SIGCOMM* (2011), ACM.

[9] Bergstra, J., Bardenet, R., Bengio, Y., and Kegl, B. Algorithms for Hyper-Parameter Optimization. In *NIPS* (2011).

[10] Bird, D. L., and Munoz, C. U. Automatic Generation of Random Self-checking Test Cases. *IBM Systems Journal* (1983).

[11] Bottou, L., and Cun, Y. L. Large Scale Online Learning. In *NIPS* (2003).

[12] Burges, C. J. From RankNet to LambdaRank to LambdaMART: An Overview.

[13] Cao, Z., Tarasov, V., Tiwari, S., and Zadok, E. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *ATC* (2018), USENIX.

[14] Chen, Q., Wang, H., Li, M., Ren, G., Li, S., Zhu, J., Li, J., Liu, C., Zhang, L., and Wang, J. SPTAG: A Library for Fast Approximate Nearest Neighbor Search. http://github.com/microsoft/SPTAG, 2018.

[15] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI* (2018), USENIX.

[16] CORTEZ, E., BONDE, A., MUZIO, A., RUSSINOVICH, M., FONTOURA, M., AND BIANCHINI, R. Resource Central: Understandingand Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP* (2017), ACM.

[17] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI* (2017), USENIX.

[18] DOMHAN, T., SPRINGENBERG, J. T., AND HUTTER, F. Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curve. In *IJCAI* (2015).

[19] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. Tech. rep., IBM Research, 2014.

[20] FRIEDMAN, J. H. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* (2001).

[21] GABEL, M., SCHUSTER, A., BACHRACH, R.-G., AND BJORNER, N. Latent Fault Detection in Large Scale Services. In *DSN* (2012), IEEE.

[22] GANGER, G. R., STRUNK, J. D., AND KLOSTERMAN, A. J. Self-* Storage: Brick-based Storage with Automated Administration. Tech. rep., CMU, 2003.

[23] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A Service for Black-Box Optimization. In *SIGKDD* (2017), ACM.

[24] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP* (2011), ACM.

[25] HASHEMI, M., SWERSKY, K., SMITH, J. A., AYERS, G., LITZ, H., CHANG, J., KOZYRAKIS, C., AND RANGANATHAN, P. Learning Memory Access Patterns. *CoRR* (2018).

[26] HUTTER, F., HOOS, H., AND LEYTON-BROWN, K. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION* (2011), Springer.

[27] JIANG, W., HU, C., PASUPATHY, S., KANEVSKY, A., LI, Z., AND ZHOU, Y. Understanding Customer Problem Troubleshooting from Storage System Logs. In *FAST* (2009), USENIX.

[28] KRASKA, T., BEUTEL, A., CHI, E. H., DEAN, J., AND POLYZOTIS, N. The Case for Learned Index Structures. In *SIGMOD* (2018), ACM.

[29] LAPTEV, N., AMIZADEH, S., AND FLINT, I. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *KDD* (2015), ACM.

[30] LI, L., JAMIESON, K., DESALVO, G., ROSTAMIZADEH, A., AND TALWALKAR, A. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. In *ICML* (2018).

[31] LI, Z. L., LIANG, C.-J. M., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly Optimizing Tail Latencies of Cloud Systems. In *ATC* (2018), USENIX.

[32] LIANG, C.-J. M., LANE, N. D., BROUWERS, N., ZHANG, L. L., KARLSSON, B., LIU, H., LIU, Y., TANG, J., SHAN, X., CHANDRA, R., AND ZHAO, F. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *MobiCom* (2014), ACM.

[33] LIANG, C.-J. M., XUE, H., YANG, M., AND ZHOU, L. The Case for Learning-and-System Co-design. In *SIGOPS Operating Systems Review* (2019), ACM.

[34] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural Adaptive Video Streaming with Pensieve. In *SIGCOMM* (2017), ACM.

[35] MIRHOSEINI, A., GOLDIE, A., PHAM, H., STEINER, B., LE, Q. V., AND DEAN, J. A Hierarchical Model for Device Placement. In *ICLR* (2018).

[36] MIRHOSEINI, A., PHAM, H., LE, Q. V., STEINER, B., LARSEN, R., ZHOU, Y., KUMAR, N., NOROUZI, M., BENGIO, S., AND DEAN, J. Device Placement Optimization with Reinforcement Learning. *CoRR* (2017).

[37] MITRA, C. R. D. J. G. G. B., AND TIWARY, S. Optimizing Query Evaluations using Reinforcement Learning for Web Search. In *SIGIR* (2018), ACM.

[38] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., AND STOICA, I. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI* (2018), USENIX.

[39] NEAMTIU, I., AND DUMITRAS, T. Cloud Software Upgrades: Challenges and Opportunities. In *MESOCA* (2011), IEEE.

[40] PARNAS, D. On the Criteria To Be Used in Decomposing System into Modules. In *ACM Communication* (1972), ACM.

[41] PATTERSON, D. A. Technical Perspective: The Data Center Is The Computer. *ACM Communication* (2008).

[42] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *ICSE* (2011), ACM.

[43] RING, M. B. CHILD: A First Step Towards Continual Learning. In *Machine Learning* (1997), Springer.

[44] RYZHOV, I. O. On the Covergence Rates of Expected Improvement Methods. In *Operations Research* (2014).

[45] STOICA, I., SONG, D., POPA, R. A., PATTERSON, D. A., MAHONEY, M. W., KATZ, R. H., JOSEPH, A. D., JORDAN, M., HELLERSTEIN, J. M., GONZALEZ, J., GOLDBERG, K., GHODSI, A., CULLER, D. E., AND ABBEEL, P. A Berkeley View of Systems Challenges for AI. Tech. rep., Berkeley, 2017.

[46] TORRALBA, A., AND EFROS, A. A. Unbiased Look at Dataset Bias. In *CVPR* (2011).

[47] WANG, J., AND LI, S. Query-driven Iterated Neighborhood Graph Search for Large Scale Indexing. In *SIGMM* (2012), ACM.

[48] WANG, M., CUI, Y., WANG, X., XIAO, S., AND JIANG, J. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network* (2018).

[49] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., YANG, F., AND ZHOU, L. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI* (2018), USENIX.

[50] XU, T., JIN, L., FAN, X., ZHOU, Y., PASUPATHY, S., AND TALWADKE, R. Hey, You Have Given Me Too Many Knobs. In *FSE* (2015), ACM.

[51] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI* (2009), USENIX.

[52] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP* (2011), ACM.

[53] ZHANG, L. L., LIANG, C.-J. M., LIU, Y., AND CHEN, E. Systematically Testing Background Services of Mobile Apps. In *ASE* (2017), ACM.

[54] ZHANG, S., AND ERNST, M. D. Which Configuration Option Should I Change? In *ICSE* (2014), ACM.

[55] ZHANG, X., LIN, Q., XU, Y., QIN, S., ZHANG, H., QIAO, B., DANG, Y., YANG, X., CHENG, Q., CHINTALAPATI, M., WU, Y., HSIEH, K., SUI, K., MENG, X., XU, Y., ZHANG, W., SHEN, F., AND ZHANG, D. Cross-dataset Time Series Anomaly Detection for Cloud Systems. In *ATC* (2019), USENIX.

Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training

Hongyu Zhu[†], *Amar Phanishayee*[⋆], *Gennady Pekhimenko*[†]
[†]*University of Toronto & Vector Institute* [⋆]Microsoft Research

Abstract

Modern deep neural network (DNN) training jobs use complex and heterogeneous software/hardware stacks. The efficacy of software-level optimizations can vary significantly when used in different deployment configurations. It is onerous and error-prone for ML practitioners and system developers to implement each optimization separately, and determine which ones will improve performance in their own configurations. Unfortunately, existing profiling tools do not aim to answer predictive questions such as "How will optimization X affect the performance of my model?". We address this critical limitation, and proposes a new profiling tool, Daydream, to help programmers efficiently explore the efficacy of DNN optimizations. Daydream models DNN execution with a fine-grained dependency graph based on low-level traces collected by CUPTI [49], and predicts runtime by simulating execution based on the dependency graph. Daydream maps the low-level traces using DNN domain-specific knowledge and introduces a set of graph-transformation primitives that can easily model a wide variety of optimizations. We show that Daydream is able to model most mainstream DNN optimization techniques and accurately predict the efficacy of optimizations that will result in significant performance improvements.

1 Introduction

Recent years have witnessed the co-evolution of deep neural network (DNN) algorithms and the underlying hardware and software design. ML researchers have developed many important models [20, 26, 27, 73] at a rapid pace, creating a huge demand for computation power [69]. To meet the demand for fast DNN computation, computer architects respond with new, AI-optimized GPUs (e.g., NVidia Turing architecture [56]) and various domain-specific hardware accelerators from FPGAs (e.g., Microsoft Catapult [64]) to ASICs (e.g., Google TPU [34], Amazon Inferentia [70]). However these accelerators might not be effective in improving performance without proper software optimizations across the full systems stack [84]. As a result, systems researchers have proposed many optimizations, targeting different bottlenecks across the system stack – for example, improving memory utilization [29, 67], better overlapping of communication with computation [25, 30, 83], and increasing communication efficiency [16]. Moreover, researchers have also developed workload-centric optimizations to exploit the stochastic nature of DNN computation. For example, precision reduction [18, 23, 42] aims to reduce runtime as well as memory consumption, and gradient compression [40, 41] aims at reducing the communication overhead in distributed training.

Despite these advances, the benefits of many proposed optimizations cannot be fully exploited due to two main reasons. First, the efficacy of many proposed performance optimizations can drastically change when applied to different ML models and deployment configurations. The hardware deployments that practitioners use might be completely different from the hardware configurations used by optimization and model inventors. Differences in DNN models, accelerator type, compute capabilities, available memory, networking capabilities, and software library versions can all shift the major runtime bottlenecks. Second, it is onerous for programmers to implement and evaluate various optimizations to identify the ones that actually work for their models. As a result, it is common for users to ask *what-if* questions such as:

Why did my DNN training workload run slowly? Will optimization X improve the performance of my model? Does GPU memory capacity limit the performance of my model? Would upgrading to a faster network improve training throughput? How will my workload scale with the number of GPUs?

The central focus of this paper is to answer the following general question for DNN training workloads: *Given a model and a deployment scenario, how can we efficiently explore the efficacy of potential solutions*? Systems researchers have tried to explore the impact of different potential performance bottlenecks (e.g., CPU, network, IO) in many non-ML contexts [5, 17, 43, 59, 60, 74]. The basic approaches to explore the what-if questions are similar: decompose the workloads into atomic tasks, profile runtime statistics for each task, model the what-if question, and use simulation to estimate performance.

These systems typically address what-if questions of the form: "How does runtime change if a task T is N times (or even infinitely) faster?" [17, 60]. Such questions can be simply modeled by shrinking task runtime. While this basic approach seems sufficient to address the central question above for ML workloads, the **diversity of DNN optimizations** introduces three key requirements unique to these workloads, thus motivating the need for a novel solution.

First, we need to **track dependencies at a kernel-level abstraction** i.e., one GPU kernel corresponds to one task (the smallest unit of execution in the dependency graph). Such fine-grained abstraction is necessary because optimizations that improve hardware utilization typically target individual compute kernels (e.g., mixed precision [42]). Meanwhile, accurate performance estimation has to consider both CPU and GPU runtime. Certain optimizations, e.g., kernel fusion, require potentially removing existing CPU and GPU tasks from the dependency graph. Existing tools do not provide such dependency tracking. It is therefore important to track kernel-level dependencies among concurrently executing tasks.

Second, we need to **map tasks to DNN layers**. In contrast to prior works that explore what-if questions in non-ML contexts, predicting the performance of DNN optimizations requires domain knowledge about DNNs to properly model them. For example, MetaFlow [33] and TASO [32] fuse DNN layers. Modeling them requires a mapping from tasks to specific DNN layers. However, collecting kernel-level traces on accelerators requires generic vendor-provided tools (e.g., NVProf [48], CUPTI [49]), which have no application specific knowledge. We therefore need to have the ability to map low-level tasks to DNN layers.

Third, we need the **ability to easily model diverse DNN optimizations**. Modeling a DNN optimization might involve not just scaling or shrinking task durations, but also complicated transformations to the dependency graph. For example, TicTac [25] reschedules communication tasks, BlueConnect [16] replaces the communication primitives to utilize parallel network channels, and the optimization proposed by Jung *et al.* [35] restructures the GPU kernel implementations. Manually manipulating the kernel-level dependency graph could be extremely intricate and error-prone. The system should enable users to flexibly and effectively model such diverse optimizations with minimal effort.

We introduce *Daydream*, a new system that fulfills all three requirements described above, and achieves our goal of answering potential what-if questions for DNN workloads. Constructing dependencies among potentially thousands of low-level tasks is not an easy problem: tasks can be spread across multiple execution threads (including both CPU threads and GPU streams), thus even for simple DNN workloads, this results in thousands of tasks to be tracked. The intricacy comes from identifying dependencies across threads. We make a key observation about DNN training workloads: despite the large number of tasks that need to be tracked, the number of concurrently executing threads is surprisingly quite limited. Based on this observation, Daydream constructs the low-level dependency graph, which provides a realistic model of overlapping among CPU, GPU, and communication runtimes in a DNN training workload. It uses a synchronization-free approach to map GPU tasks onto appropriate higher-level DNN layer abstractions. We also introduce a set of graph-transformation rules, allowing programmers to effectively model various performance optimizations. After modeling the optimization, Daydream simulates the execution based on the new dependency graph to predict the overall runtime. In our evaluation, we show that Daydream is able to distinguish effective DNN optimizations from those that will bring limited improvements by accurately predicting their performance speedups.

In summary, we make the following key contributions:

- We make the observation that fine-grained tasks in DNN training workloads are highly sequential. This greatly simplifies dependency graph construction, over thousands of tasks, as we only need to identify a limited number of inter-thread dependencies.
- Daydream introduces the abstraction of a kernel-granularity dependency graph that contains mappings back to DNN specific abstractions (layers), by collecting profiling data, instrumenting DNN frameworks, and exploiting information from vendor-provided tools like CUPTI. Daydream also provides primitives to mutate the dependency graph in the form of simple graph transformations. Taken together this enables programmers to both (i) model a diverse set of popular optimizations spanning kernel- and layer-level enhancements by using simple graph-transformation primitives, and (ii) estimate the efficacy of optimizations by simulating execution time based on optimization-induced graph mutations.
- We extensively evaluate Daydream, with *five* different optimizations on *five* DNN models across *three* distinct applications. We show that Daydream can effectively detect which optimizations provide improvements and also accurately predict their magnitude for different DNN models and deployments. For example, we estimate that using mixed precision will improve the iteration time of training BERT$_{\text{LARGE}}$ model by 17.2% (with <3% error), while the kernel fusion technique can improve it by 38.7% (with <7% error). We can also accurately predict performance in distributed training with different number of workers and variable network bandwidth, based on runtime profiles collected from a single-GPU setting.

2 DNN Training Optimizations and Tools

DNN training is an iterative algorithm, in which one iteration consists of three phases: (i) *forward*, (ii) *backward*, and (iii) *weight update*. The *forward* phase takes training data samples as input and produces output based on current weights

Optimization Goal	Strategy	Technique Examples
Improving Hardware Utilization in Single-Worker Setting	Increasing Mini-batch Size by Reducing Memory Footprints	**vDNN** [67], **Gist** [29], Chen *et al.* [14]
	Reducing Precision	***Micikevicius et al.*** [42], Gupta *et al.* [23], Das *et al.* [18]
	Fusing Kernels/Layers	***FusedAdam*** [52], **MetaFlow** [33], Ashari *et al.* [10], TASO [32]
	Improving Low-level Kernel Implementation	***Restructing Batchnorm*** [35], Tensor Comprehensions [72], Kjolstad *et al.* [37], TVM [13]
Lowering Communication Overhead in Distributed Training	Reducing Communication Workloads	**Deep Gradient Compression** [40], AdaComm [76], Parallax [36], TernGrad [78], QSGD [8]
	Improving Communication Efficiency/Overlap	***Wait-free Backprop*** [83], ***P3*** [30], **BlueConnect** [16], TicTac [25], BytePS [62], Xue *et al.* [80]

Table 1: Representative optimizations for DNN training. We show how we can accurately estimate the performance of optimizations (shown in ***italics***) in Section 6, and can effectively model many other optimizations (shown in **bold**) in Section 5.

(or parameters). The error between the *forward* output and the input data labels is fed to the *backward* phase, which computes the gradients of weights with respect to the input data. The *weight update* phase then uses the gradients to update weights accordingly. In each iteration, the input data samples are randomly selected [11], forming a *mini-batch* of input.

2.1 DNN Training Optimizations

Modern DNNs have millions of parameters [24], resulting in training times of days or even weeks [38]. To improve DNN training performance, researchers have proposed various strategies focusing on different optimization goals. To understand the potential what-if questions and how to design a system to answer them, we study a list of software-level techniques that speedup DNN training from top systems and ML conferences in recent years. Table 1 shows our summary.

Exploiting computation power of hardware accelerators. ML programmers often use large mini-batches, within the memory budget, for better hardware utilization and faster convergence. This motivates strategies that reduce the memory footprint of DNN training and hence enables training with larger mini-batch sizes [14, 29, 67]. Researchers have also proposed some generic strategies to increase hardware utilization, including precision reduction [18, 23, 42], kernel/layer fusion [10, 32, 33], and improving low-level kernel implementation [13, 35, 37, 72]. Meanwhile, libraries such as cuDNN [15], cuBLAS [45], MKL [75], Eigen [1], and NCCL [46] are also constantly evolving to provide operations and primitives that can better utilize underlying hardware.

Scalable distributed training. Data parallelism [11] is a simple and effective strategy to improve training performance. Using multiple accelerators significantly reduces DNN training time to hours or even minutes [44]. This success is mainly based on the techniques that guarantee model convergence under extremely large mini-batch size [7, 22, 81]. One of the major performance bottlenecks for distributed training is communication, which can be optimized by compressing traffic [40, 41, 76, 78], increasing network utilization [16, 80], or increasing the overlap between communication and computation [25, 30, 83]. Exploring the efficacy of these optimizations without prediction requires a multi-machine cluster. Our proposed design, Daydream, avoids the potential cost of cluster setup (i.e. extra machines, accelerators, high-speed communication), by predicting distributed training performance with profiles collected from a single-worker environment.

2.2 Profiling Tools for DNNs

As the full ML system stack is constantly evolving, profiling tools play a key role in helping programmers identify the performance bottlenecks under different system configurations.

Hardware profiling tools. Modern DNN training heavily relies on hardware accelerators such as GPUs [56] and TPUs [34]. To help programmers develop highly efficient applications, hardware vendors provide profiling tools that can expose hardware performance counters. For example, NVProf [48] provides programmers with information including start/end time, core utilization, memory throughput, cache miss rate, along with hundreds of other hardware counters for every GPU kernel. CUPTI [49] enables programmers to extract and manipulate these counters at runtime. Nsight [47] aims to provide details on the state of more fine-grained counters for recent GPU architectures [56]. Our proposed system, Daydream, relies on CUPTI to collect low-level traces for further analysis.

Framework built-in tools. For more intuitive profiling results, it is often desirable for a profiler to show runtime statistics for framework operations, or even DNN layers. DNN frameworks have built-in tools to achieve this goal by correlating the hardware counters with runtime information collected in frameworks. TensorFlow [3], coupled with the Cloud TPU Tool [21], can provide an execution timeline and runtime statistics for each TensorFlow operation. Similarly, other mainstream frameworks (e.g., MXNet [12] and PyTorch [61]) provide built-in tools that can extract per-layer or per-operation runtime from both the CPU and the GPU. The framework built-in tools render intuitive results for pro-

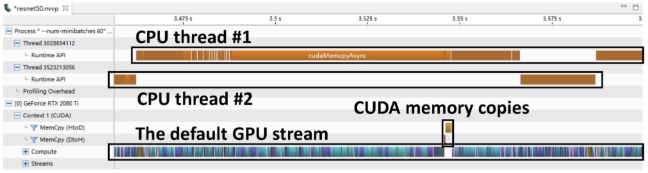


Figure 1: NVProf timeline example of training ResNet-50.

grammers, but omit important details (for example, the CPU runtime). We show in our work that such information is crucial in building an accurate runtime predictor.

3 Key Ideas

In this section we highlight the key ideas and observations behind the Daydream design.

Constructing kernel-granularity dependency graph. The neural network topology is a natural graph structure in which nodes are DNN operators or layers. Most mainstream DNN frameworks [12, 61] provide built-in tools to record the layer-level runtime profile. The layer-level abstraction is intuitive for programmers to understand the "where time goes" question, but hides important information about the parallel execution of the CPU functions, GPU kernels, and memory transfers. This information is crucial for accurate performance predictions. For example, optimizations that reduce numerical precision will change the duration of GPU kernels while the CPU runtime remains unchanged, and optimizations like vDNN [67] will inject CUDA memory copies, without changing the duration of GPU kernels. It is extremely hard to predict how duration of each layer changes when applying these optimizations if lacking low-level details about CPU and GPU runtime. To accommodate optimizations that target fine granularity tasks (such as GPU kernels), our proposed system, Daydream chooses to model the training workloads using a kernel-level dependency graph (i.e., each GPU kernel has one corresponding task in the graph), incorporating detailed traces of CPU, GPU and communication runtime.

With a large number of kernel-level tasks that are spread across several threads and CUDA streams, the complexity of constructing the dependency graph comes mainly from identifying the inter-thread dependencies [74]. Existing tools do not provide such dependency tracking. We make the following key observations about the DNN training workloads to overcome this general challenge of dependency tracking in concurrent systems. First, for the implementations in the mainstream frameworks [12, 61], once a mini-batch has been prepared by data loading threads, only one or two CPU threads are involved in the control flow of computation.[1] Second, there is a very limited number of concurrent GPU kernels. Such serialization of GPU kernels is due to two main reasons: (i) GPU kernels in the modern cuDNN library achieve high GPU core utilization; (ii) ML frameworks usually invoke only one CUDA stream. Figure 1 shows the NVProf profiles of one training iteration of ResNet-50. There are two CPU threads involved, but no CPU tasks run concurrently. The high serialization of low-level traces is not a unique phenomenon for just convolutional networks. We observe a similar phenomenon in most DNN training workloads.

Based on these insights, Daydream constructs the kernel-level dependency graph in three major steps. First, Daydream uses CUPTI to extract traces of all GPU kernels, CUDA memory copies, and CUDA APIs. Second, Daydream captures the dependencies between CPU and GPU tasks, caused by CUDA synchronizations and GPU kernel launches. Third, when predicting performance for distributed training, Daydream adds communication tasks to the dependency graph.

Synchronization-free task-to-layer mapping. In distributed training, mainstream frameworks implement the wait-free backpropagation strategy [83] to overlap communication with computation. This strategy immediately transfers gradients once they are computed by corresponding backward layers. To properly add dependencies related to communication tasks, we need the task-to-layer mapping to know when the computation of each layer ends. Meanwhile, accurately modeling DNN optimizations by changing the graph potentially requires this task-to-layer mapping to determine which tasks are involved and how to change them.

Unfortunately, vendor-provided tools like CUPTI do not have the required knowledge about these applications and building such a mapping requires extra DNN framework instrumentation. A naïve approach to achieve this mapping is to compare the start and stop timestamps of GPU kernels and DNN layers. This requires additional CUDA synchronization calls for each layer since GPU kernels are launched asynchronously. However, such synchronizations might significantly alter the execution runtime by adding additional dependencies from GPU to CPU tasks. Hence, we design a synchronization-free procedure to achieve this mapping by instrumenting timestamps for each layer in the frameworks, and utilizing the correlations between CPU and GPU tasks.

Representing complex optimizations with simple graph-transformation primitives. As shown in Table 1, DNN optimizations target a wide range of performance bottlenecks with various approaches. Unlike prior dependency graph analysis in non-ML contexts [17, 59, 60], where users can model most what-if questions by simply shrinking and scaling task runtime, accurately modeling DNN optimizations with the low-level dependency graph might require complicated changes to the dependency graph. Manually changing the kernel-level graph to model optimizations could be both complicated and error-prone, and the programmers might simply opt to rather directly implement the optimizations.

To address this problem, we propose a small set of graph-transformation primitives, so that popular optimization techniques can be effectively represented as a combination of

[1] Our approach can be generalized to frameworks that use more concurrent CPU threads.

these primitives. These primitives include (i) task insertion/removal, (ii) task selection and update, and (iii) changing the policy for scheduling tasks. The proposed primitives are simple yet powerful enough to represent many different optimizations as we will show in Section 5. They play a key role in realizing our goal of efficiently exploring what-if questions.

In summary, Daydream introduces the abstraction of a kernel-granularity dependency graph that contains mappings back to DNN specific abstractions (layers). It tracks dependencies by collecting profiling data as well as instrumenting DNN frameworks. Daydream also provides primitives to mutate the dependency graph in the form of simple graph transformations. Altogether this enables programmers to both (i) model a diverse set of popular optimizations spanning kernel- and layer-level enhancements by using simple graph-transformation primitives, and (ii) estimate the efficacy of optimizations by simulating execution time based on optimization-induced graph mutations.

4 Design

We describe Daydream's design with an emphasis on how to construct Daydream's proposed graph abstraction: the kernel-granularity dependency graph with mappings back to DNN layers. We also describe the primitives for mutating this graph to model different optimizations and how Daydream uses the graph to estimate the efficacy of various DNN optimizations.

4.1 Overview of Daydream

Figure 2 shows the workflow of performance prediction in Daydream. It consists of the following four phases:

Phase 1: Trace collection. Constructing a kernel-level dependency graph requires low-level details for all tasks. These details are extremely massive, differ across ML frameworks, and can be obtained by profiling a baseline workload. Daydream collects low-level profiling data using CUPTI [49], a tool which provides details for all CPU/GPU tasks including name, start time, duration, CUDA stream ID, thread ID, etc. We manually augment three popular frameworks (Caffe, MXNet, PyTorch) for use with CUPTI and modify the layer modules of these frameworks to collect timestamps of each layer, which will be used for task-to-layer mapping, described in Section 4.3. Through our instrumentation, we also collect the necessary information (e.g., size of gradients) to construct the dependency graph of distributed training via a profile collected in a single worker setting.

Phase 2: Dependency graph construction. Daydream constructs the dependency graph with details of tasks provided by the first phase. A dependency could be induced by domain knowledge (e.g., a GPU task triggers a communication task), or by hardware/software implementation (e.g., a cudaLaunchKernel API triggers the corresponding GPU task). Based on our analysis, we identify five different types of dependencies (described in Section 4.2.2), which are sufficient for Daydream to accurately simulate baseline execution.

Phase 3: Graph transformation. To estimate the efficacy of a given optimization, Daydream models the optimization by transforming the dependency graph. Daydream provides a set of primitives (e.g. selection, insertion/removal) to represent these transformations. We design these primitives in a way such that they are succinct (easy to use), flexible (able to depict a wide range of optimizations), and accurate (being able to achieve high prediction accuracy).

Algorithm 1: Daydream's Simulation Algorithm

```
Input  : Dependency graph: $G(V,E)$
Output : The start time of each task $u \in V$
$F \leftarrow \emptyset$ // initialize the frontier task set
$P \leftarrow \{0\}$ // initialize thread progress
foreach task $u \in V$ do
    $u.ref \leftarrow |\{u'sparents\}|$
    if $u.ref = 0$ then
        $F \leftarrow F \cup \{u\}$
end
while $F \neq \emptyset$ do
    $u \leftarrow schedule(F)$ // pick a task to exec.
    $t \leftarrow u.ExecutionThread$
    $F \leftarrow F - \{u\}$
    $u.start \leftarrow max(P[t], u.start)$
    $P[t] \leftarrow u.start + u.duration + u.gap$
    foreach $c \in u.children$ do
        $c.ref \leftarrow c.ref - 1$
        $c.start \leftarrow$
          $max(c.start, u.start + u.duration + u.gap)$
        if $c.ref = 0$ then
            $F \leftarrow F \cup \{c\}$
        end
    end
end
```

Phase 4: Runtime simulation. Daydream simulates the execution of optimizations to predict runtime based on the dependency graph. Algorithm 1 shows the simulation process, which traverses the dependency graph and puts tasks into execution threads. In each iteration, Daydream picks one task from the execution frontier (i.e. tasks that are ready to execute), dispatches it to its corresponding execution thread, and updates the thread progress. The simulation determines the start time of each task and records the total execution time.

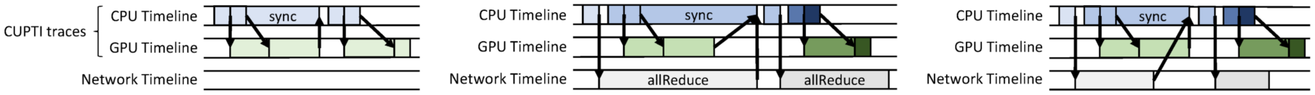
4.2 Dependency Graph Construction

Constructing the dependency graph is essential to determine the node (task) set and edge (dependency) set.

4.2.1 Task

Daydream's kernel-level dependency graph contains the following four types of tasks:

GPU tasks. Each GPU task in the graph corresponds to one GPU kernel. Daydream also views CUDA memory copies as



(a) Constructing the dependency graph based on CUPTI traces (the black arrows represent task dependencies).

(b) Mapping each task to DNN layers (shown in different colors in the figure), then inserting communication tasks based on mapping and instrumentation.

(c) Predicting "what if network bandwidth is 2×" by shrinking allReduce duration by 2× and simulating the new dependency graph.

Figure 2: An example showing Daydream's overall workflow for predicting runtime assuming network bandwidth doubles.

GPU tasks, because each memory copy is associated with a specific CUDA stream, and therefore has dependencies with other GPU kernels. The runtime of all these tasks can be collected using CUPTI.

CPU tasks. To model the concurrency and dependencies between CPU runtime and the GPU runtime, Daydream generates CPU tasks based on CPU traces collected by CUPTI. One of the limitations of CUPTI is that it can only expose CUDA-related traces. Instead of adding massive instrumentation to the framework, Daydream captures the non-CUDA runtime by recording the lengths of gaps between consecutive CPU tasks (shown in line 13 of Algorithm 1).

Data loading tasks. One data loading task corresponds to loading one mini-batch from disk/flash to CPU memory. We include data loading tasks for completeness, even though data loading in most DNN training workloads is not a performance bottleneck. In Daydream's implementation, we treat all data loading tasks as CPU tasks.

Communication tasks. A communication task corresponds to one communication primitive, e.g., a push/pull operation in parameter-server based frameworks [39], or an all-reduce operation in decentralized frameworks. When predicting distributed training performance, Daydream automatically adds communication tasks to the dependency graph based on a single-worker profile. We notice that in PyTorch, gradients from multiple layers can be grouped and sent with a single allReduce primitive [2]. Thus, properly adding communication tasks to a PyTorch profile requires additional instrumentation to extract knowledge about gradients grouping.

Given the types of tasks in the graph, Daydream collects and maintains the following information for each task, which is later used in what-if analysis and simulation:

ExecutionThread. Depending on the type of a task, its execution thread can be on of the following: (i) a CPU process, (ii) a GPU stream, and (iii) a communication channel. A data loading task is executed in a CPU process. A CPU process has a process ID, a GPU stream has a stream ID, and a communication channel could be send/receive when using parameter server primitives, or a unified one when using collective primitives. This field is used in line 10 of Algorithm 1.

Duration. This field specifies how long a task takes to execute. The duration of a CPU/GPU task is collected by CUPTI. The runtime of data loading tasks is measured by injecting timestamps to the framework. Daydream aims to predict distributed training performance based on profiling in a single-GPU configuration. Hence we calculate the duration of all communication task based on the size of gradients, the communication type (push/pull/all-reduce), and the network bandwidth. These numbers can be obtained based on knowledge of the DNN model and framework implementation.

Gap. The duration of low-level CUDA APIs (e.g., `cudaMalloc`) might be only tens of microseconds, which is of the same magnitude as the runtime of their non-CUDA equivalent C functions (e.g., `malloc`), or the runtime of the call stack from Python front-end to C back-end. NVidia-provided tools cannot expose non-CUDA traces, but they are indispensable to simulation accuracy. The non-CUDA CPU runtime is usually not a target for optimization in DNN models, hence, we do not need to define and measure corresponding tasks. Instead, for each CPU task in our current definition, we measure the gap between its end and the start of the next task in the same execution thread, and simulate these gaps in Algorithm 1.

Layer. This field refers to which DNN layer a task belongs to, which is necessary information for programmers to transform the graph and model optimizations. Daydream uses a synchronization-free approach to map a task to DNN layers. We will describe the details of this approach in Section 4.3.

4.2.2 Dependency

Based on our discussion in Section 3, we identify the following five types of dependencies for accurate simulations.

Sequential order of CPU tasks in the same thread. CPU tasks in the same thread are serialized. The order that CPU tasks are executed in is determined by the framework and does not change in two separate executions. We add a dependency between each two consecutive CPU tasks in the same thread.

Sequential order of GPU tasks in the same CUDA stream. GPU kernels belonging to the same CUDA stream are executed sequentially. Similar to CPU tasks, the order of GPU tasks in the same stream does not change between executions. Hence, two consecutive GPU tasks in the same CUDA stream have a dependency between them.

Correlation from CUDA APIs to GPU kernels. Each GPU kernel or CUDA memory copy has a corresponding CPU-sided CUDA API (`cudaLaunch`, `cudaMemcpy`, or `cudaMemcpyAsync`) that triggers the GPU task. CUPTI provides a correlation ID for every CUDA API and GPU kernel. A GPU kernel is dependant on a CUDA API if they share the same correlation ID.

CUDA Synchronization. A CUDA synchronization API

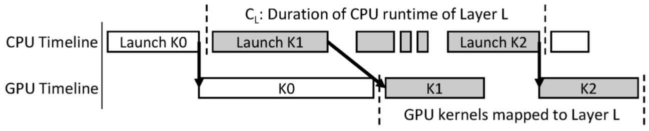

C_L: Duration of CPU runtime of Layer L
CPU Timeline
Launch K0
Launch K1
Launch K2
GPU Timeline
K0
K1
K2
GPU kernels mapped to Layer L

Figure 3: The mapping of GPU kernels to a layer. CUPTI provides correlations between CUDA launches and GPU kernels.

(e.g., `cudaDeviceSynchronize`) is invoked on CPU, and returns after GPU kernels (or CUDA memory copies) that are launched before this synchronization complete. A CUDA synchronization therefore generates dependency from a GPU task to a CPU task. Similar to CUDA synchronizations, even though a `cudaMemcpyAsyncDtoH` call returns before a memory copy completes, we found it still blocks the CPU until all previous GPU kernels on the same stream are completed.

Communication. Mainstream frameworks including PyTorch and MXNet implement the wait-free backpropagation strategy [83] to schedule gradient communication. Here, a communication primitive is launched as soon as the weight gradients are ready, thus overlapping communication with the backward phases of subsequent layers. Hence, we need to know the runtime of DNN layers (not just kernels) to determine which tasks trigger communication.

4.3 Mapping Tasks to Layers

The task-to-layer mapping enables Daydream to construct the dependency graph for distributed training, and provides necessary domain knowledge for Daydream to model DNN optimizations. Figure 3 shows how Daydream determines which tasks belong to a certain layer. Let L be the forward phase of a DNN layer. Daydream collects the CPU and GPU runtime information using CUPTI [49], as well as timestamps before and after the forward, backward, and weight update phases for each layer. The start and end timestamps of L will determine the CPU runtime of L (denoted by C_L). To determine the GPU runtime of L, Daydream gathers all CUDA launch calls invoked during C_L. With CUPTI providing the correlations between CUDA launch calls and corresponding GPU kernels, Daydream can identify all the GPU kernels launched during C_L, and map these kernels to L. This process can also be applied to the backward or weight update phases of any layers, and can be further generalized to any code region of interest in the framework or user-level programs.

4.4 Graph Transformation

What-if analysis by transforming the graph and simulating the execution requires input about the optimizations from programmers. Daydream provides a set of primitives for programmers to model DNN optimizations by modifying the graph. Like most what-if analysis in non-ML contexts, modeling DNN optimizations requires potentially shrinking or scaling the duration of tasks (the `shrink/scale` primitives). We carefully study common DNN optimization techniques and identify the following primitives (besides the `shrink/scale` primitives), which are sufficient for programmers to describe those optimizations.

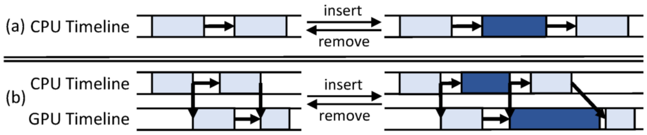


Figure 4: Insert/Remove a (a) CPU task; (b) GPU task.

Insert/Remove a task. Inserting a task to an execution thread just involves an appending of a node to a linked list. Figure 4 shows how this process works. When inserting a GPU task, we need to insert the corresponding CPU tasks that launch it. Which CPU tasks to insert and their duration depend on the framework implementation, and can be inferred based on collected traces.

Select. This operation allows users to select tasks of interest for further operations. One potentially useful selection criterion is select-by-layer, as many optimizations are depicted based on DNN layers. Another potentially useful criterion is to select by keywords in task names, based on knowledge of the software library (e.g., cuDNN [50]). For example, kernels with keywords such as `elementwise` or `PointwiseApply` in the names are element-wise arithmetic operations. These kernels are typically *not* compute-bound, and could be much shorter than their corresponding CUDA launch calls. Similarly, kernels with `sgemm` string in names are compute-bound matrix-multiplications.

Schedule. The `schedule` function picks one task from a set of frontier tasks that are ready to execute (line 9 in Algorithm 1). By default, it picks the task with the earliest start. Programmers can override this function and implement any custom scheduling policy. which is useful to model optimizations that increase computation-communication overlap.

5 Modeling Optimizations

To demonstrate that Daydream is able to estimate the performance of the most common optimizations in DNN training, we select ten techniques from Table 1 with different optimization goals. We show that we can easily model these optimizations using the primitives Daydream provides. [2]

5.1 Optimizations for Evaluation

We select the following five DNN optimizations, which we are able to acquire the implementations, to evaluate Daydream's prediction accuracy. We use implementations from the authors of these optimizations in cases where they were not readily available.

Automatic Mixed Precision (AMP). We aim to predict the efficacy of the AMP optimization [42], implemented using NVidia's Apex package [51]. We expect that AMP will

[2] We show pseudo code for AMP in this section. Refer to our arxiv version [85] for the pseudo code of all examples shown in Section 5.

improve memory-bounded GPU kernels by 2× because the number of transferred bits is halved. With Tensor Cores in the Volta and Turing architectures, AMP empirically yields up to 3× speedup on the most compute-intensive workloads [58]. To predict AMP performance, we simply `select` all the compute-intensive (e.g., `sgemm`, `conv`) kernels and memory-bounded (e.g., `elementwise`, `batchnorm`, `RELU`) kernels, and `shrink` their duration by 3× and 2× respectively. We show the pseudo code for modeling AMP in Algorithm 3.

Algorithm 2: What_If_AMP

```
Input  : Dependency graph: $G(V,E)$
Output : A modified graph $G(V,E)$ to model AMP
$GPUTasks \leftarrow \{G.Select(funcPtr(IsOnGPU))\}$
foreach $u \in GPUTasks$ do
    if "sgemm" in u.Name or "scudnn" in u.Name then
        $u.duration \leftarrow u.duration/3$
    else
        $u.duration \leftarrow u.duration/2$
    end
end
```

FusedAdam Optimizer. We use the FusedAdam optimizer [52] implemented in NVidia's Apex package [51] as an example for the kernel fusion optimization. This optimizer fuses all kernels in one weight update phase into one unified kernel. It is applicable to the models that use the Adam optimizer (e.g., GNMT, BERT). Daydream uses the kernel-to-layer mapping to identify the CPU/GPU tasks that belong to a weight update phase. We `remove` all these tasks, then `insert` a new GPU task whose duration is roughly estimated by the sum of all removed compute-intensive kernels.

Reconstructing Batchnorm. Recently Jung et al. [35] proposed a technique that optimizes non-convolutional layers in state-of-the-art CNNs. It first splits each batch normalization layer into two sub-layers, then fuses the first sub-layer with the previous convolutional layer, and the second sub-layer with the following activation and convolutional layers. We `remove` the affected activation kernels when estimating performance, since they are memory-bound kernels now fused with compute-intensive convolutional kernels. For the batch nomalization layers, we estimate that the GPU kernels will be improved by 2× since this optimization halves the amount of input data that these layers load from GPU memory.

Distributed Training. Using Daydream we can accurately predict distributed training performance with the profile based on the single-GPU environment. We evaluate Daydream's prediction based on PyTorch, which uses collective communication primitives from the NCCl library [46]. PyTorch groups gradients from multiple layers into buckets before transferring them. Hence, to predict distributed training performance, we need to `insert` one allReduce task for every bucket. The dependencies of the inserted tasks are determined based on the layer-to-bucket mapping (which requires additional instrumentation to the PyTorch framework).

Priority-Based Parameter Propagation (P3). P3 [30] is a technique that optimizes communication overhead by slicing and prioritizing. We evaluate Daydream's prediction of P3 based on MXNet, which uses the parameter-server mechanism [39]. In order to model parameter slicing, we `insert` multiple push task and pull tasks between the backward and the forward GPU tasks for each layer. The duration of the push/pull task is calculated from the slice size and the network bandwidth. To model the priority scheduling, we override the `schedule` function with a priority queue.

5.2 Modeling Additional Optimizations

In addition to the above optimizations, we show that Daydream is capable of modeling an additional set of diverse DNN optimizations.

BlueConnect. BlueConnect [16] optimizes communication by decomposing the allReduce primitives into a series of reduce-scatter and all-gather primitives. These primitives run concurrently as they use parallel communication channels. To predict the performance of BlueConnect, instead of `insert`ing regular allReduce or push/pull tasks, we need to `insert` reduce-scatter and all-gather tasks, and assign them to corresponding network channels (the duration can be estimated according to formulas shown in [57]).

MetaFlow. MetaFlow [33] is a layer-fusion technique to optimize DNN training by fusing DNN layers to simplify the DNN topology. We `select` the GPU kernels of substituted layers, `remove` them, and `insert` GPU kernels of new layers to predict the performance of MetaFlow in Daydream. The new layers are mostly existing layers with different dimensions; their GPU kernel durations can be inferred by profiling.

vDNN. Virtualized DNN [67] reduces GPU memory consumption by temporarily offloading intermediate data from GPU memory to CPU memory. The offloaded data needs to be prefetched back to GPU to perform execution, which causes potential performance overhead due to PCIe traffic or late prefetching. To predict the performance overhead using Daydream, we only need to `insert` additional CUDA memory copies, and override the `schedule` function to implement a custom prefetching policy.

Gist. Gist [29] reduces GPU memory consumption by storing encoded intermediate data and decoding before the data is used. The encoding and decoding introduces performance overhead. We `insert` extra encoding and decoding GPU kernels (along with `cudaLaunchKernel` calls in CPU) to estimate the performance overhead in Daydream. The duration of the inserted encoding/decoding kernels can be estimated using existing element-wise kernels.

Deep Gradient Compression (DGC). DGC [40] is a technique that reduces communication overhead by compressing the gradients. To estimate performance, we: (i) `scale` the duration of communication; (ii) `insert` the GPU tasks of compression and decompression. The duration of inserted

Application	Model	Dataset
Image Classification	VGG19 [71] DenseNet-121 [28] ResNet-50 [27]	ImageNet [19]
Machine Translation	GNMT [79]	WMT16 [4]
Language Modeling	BERT [20]	SQuAD [65]

Table 2: The models and datasets we use in this paper.

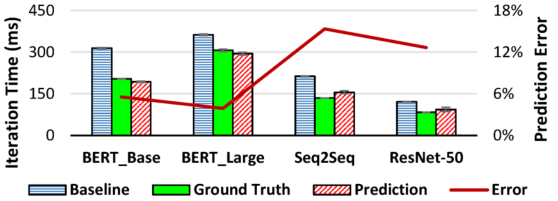


Figure 5: AMP – comparing baseline (FP32), ground truth with mixed precision, and predictions by Daydream.

GPU tasks can be estimated according to the compression rate and duration of existing element-wise GPU kernels.

6 Evaluation

6.1 Methodology

We implement Daydream based on three mainstream DNN frameworks: PyTorch [61], MXNet [12], and Caffe [31]. We add CUPTI [49] support to each framework to obtain traces of CUDA APIs and GPU kernels. We also add instrumentation to the frameworks to acquire layer-wise timestamps for the kernel-to-layer mapping process, and communication information such as the size of each allReduce call and their dependencies with other layer-wise computation.

Infrastructure. We evaluate Daydream's runtime prediction on a cluster of four machines. Each machine contains one AMD EPYC 7601 16-core processor [9], and four 2080Ti GPUs [55] with 11GB GDDR6 memory each, connected through PCIe 3.0 [6]. Our experiments are based on Ubuntu 16.04, CUDA v10.0 [53], cuDNN v7.4.1 [54], and NCCL v2.4.2 [46]. Our software implementation is based on PyTorch v1.0, MXNet v1.1, and Caffe v1.0.

Models. Table 2 shows the DNN models and datasets we use to evaluate Daydream. We select five DNN models from three different applications, covering a diverse set of DNN models. For the BERT model, we evaluate both "base" and "large" versions. The difference between these versions is that the "base" version contains 12 "Transformer blocks" (the main layer type in BERT) where as the "large" version contains 24.

6.2 Automatic Mixed Precision (AMP)

We evaluate Daydream's prediction accuracy of AMP [42], which is implemented in NVidia's Apex package [51] based on the PyTorch framework. Figure 5 shows the performance of using AMP and the corresponding performance prediction given by Daydream. Our predictions have errors below 13% for all the models we evaluate.

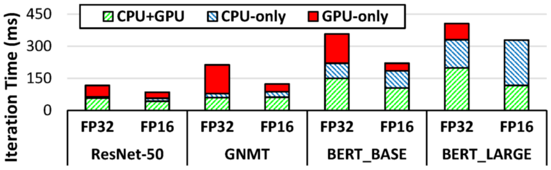


Figure 6: Runtime breakdown of the baseline (FP32) and mixed precision (FP16).

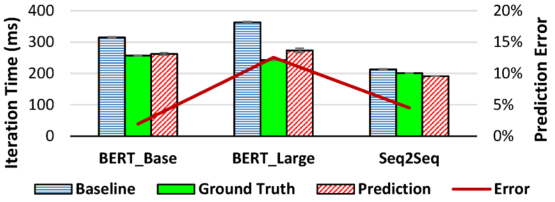


Figure 7: FusedAdam - comparing baseline (FP32), ground truth with FusedAdam, and predictions by Daydream.

Our experiments show that using AMP brings speedups generally less than 2× – much less than the theoretical boost of using AMP for individual kernels (e.g., 3×). To understand how AMP improves performance, we break down the overall runtime into the following three components:

CPU-only runtime. This component refers to the runtime when the CPU is busy, but the GPU is not executing any kernels. It is straightforward to calculate this runtime by simply subtracting all GPU kernel runtime from the total runtime.

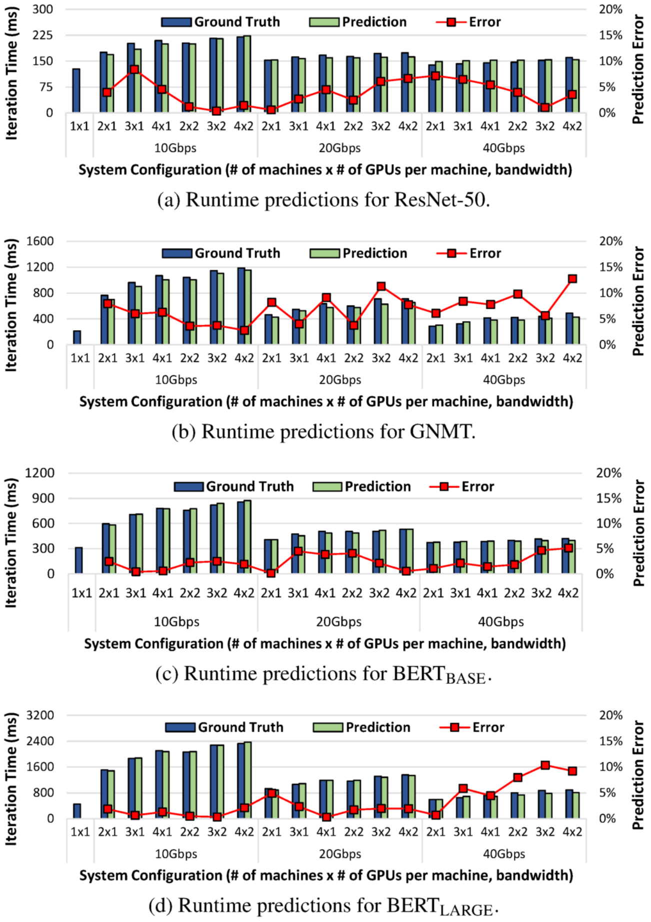
GPU-only runtime. This component refers to the runtime when the CPU is waiting for the GPU kernels to complete. It includes not only the duration of CUDA synchronization APIs, but also the `cudaMemcpyAsync` calls of all the device-to-host CUDA memory copies.

CPU+GPU parallel runtime. This component refers to the runtime when both CPU and GPU are busy. We calculate this part of runtime by deducting the CPU-only and GPU-only parts from the total runtime.

Figure 6 shows the runtime breakdown of the models we evaluated. CPU runtime generally becomes the new performance bottleneck in the models that incur limited speedups (e.g., $\text{BERT}_{\text{LARGE}}$). When applying AMP, the CPU bottleneck increases, because the GPU runtime becomes shorter and part of the CPU+GPU parallel runtime is shifted to the CPU-only runtime. The overall runtime improvement comes mostly from the reduction of GPU-only runtime while CPU runtime barely changes. This demonstrates the necessity of the kernel-level abstraction when predicting performance.

6.3 FusedAdam Optimizer

We apply the FusedAdam optimization to the BERT and GNMT models as they use the Adam optimizer. Figure 7 shows the performance of using the FusedAdam optimizer. Our predictions are within 13% of the ground truth runtime.



(a) Runtime predictions for ResNet-50.

(b) Runtime predictions for GNMT.

(c) Runtime predictions for BERT$_{\text{BASE}}$.

(d) Runtime predictions for BERT$_{\text{LARGE}}$.

Figure 8: The error between Daydream's runtime predictions and the baseline with synchronization before each allReduce under various system configurations.

There are two reasons why the FusedAdam optimizer substantially improves the performance of BERT models. First, unlike most DNN training workloads, the weight update phase is a significant proportion of a BERT model's iteration runtime (around 30% for BERT$_{\text{BASE}}$ and 45% for BERT$_{\text{LARGE}}$). Second, the weight update phase consists of very many element-wise GPU kernels (2633 for BERT$_{\text{BASE}}$, 5164 for BERT$_{\text{LARGE}}$). Thus, the CUDA launch calls on the CPU become the main bottleneck. The FusedAdam optimizer almost eliminates all CPU kernel launch overhead in the weight update phase by fusing all GPU kernels into one single GPU kernel. Compared to BERT models, the GNMT model spends less than 10% of its iteration time on the weight update phase, explaining the lower speedup improvements.

6.4 Reconstructing Batchnorm

We evaluate our performance prediction for the optimization of reconstructing batch normalization [35] based on the Caffe implementation of DenseNet-121 [28]. Using Daydream, we predict that reconstructing batchnorm will yield a moderate performance improvement of 12.7% compared to the baseline. This suggests that reconstructing batchnorm in our configuration is less promising than the paper claims (17.5% speedup). We verify this conclusion by testing the ground truth implementation of reconstructing batchnorm, and find out that this optimization yields even lower 7% speedup.

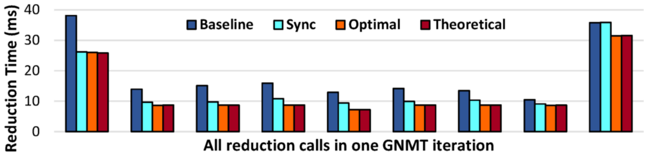


Figure 9: Comparison of all individual reduction runtimes in one training iteration of GNMT. **Baseline**: runtime measured in regular training; **Sync**: runtime measured with an additional CUDA synchronization before each reduction; **Optimal**: runtime measured when executing exclusively; **Theoretical**: runtime calculated using the formula [57].

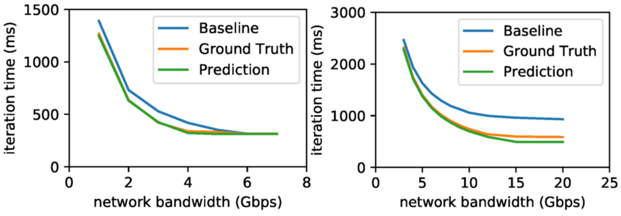
We notice that there are two main reasons for the difference between our prediction and the ground truth. First, the ground truth uses a completely new implementation of the batchnorm layers, and it is hard to precisely predict the runtime of newly implemented kernels. Second, the ground truth implementation introduces new CUDA memory copies and allocations, which add performance overhead. Obtaining a very precise estimate would require us to understand not just the high-level idea from the paper, but also the detailed implementation of the user-level programs and the Caffe framework.

6.5 Distributed Training

Next we evaluate distributed training using PyTorch with the NCCL [46] library. Figure 8 shows the comparisons between runtimes predicted by Daydream and the measured ground truth runtimes, for each DNN model under different system configurations. We evaluate the prediction accuracy for Ethernet and InfiniBand connecting multi-machine systems under different network bandwidths (10, 20, 40 Gbps). In most of the configurations, Daydream predicts distributed runtime with at most 10% prediction error, with a few exceptions for the 20Gbps and 40Gbps configurations.

The prediction errors of the overall iteration times are mainly due to inaccurate estimates of individual NCCL primitives. Figure 9 shows the comparisons of NCCL allReduce calls between the ground truths and predictions. The ground truths are on average 34% higher than the theoretical values.

An NCCL primitive is both a communication primitive and a GPU kernel, suggesting that it could be bottlenecked by two types of hardware resources: (i) the network bandwidth, and (ii) GPU resources (e.g., memory bandwidth, streaming multiprocessors). Figure 9 shows that the predicted values are very close to the runtimes measured when running NCCL primitives exclusively. This suggests that the ground truth is slower because they compete for GPU resources with other GPU kernels. Based on this insight, we try to reduce this interference by adding CUDA synchronizations before invoking NCCL



(a) ResNet-50.

(b) VGG-19.

Figure 10: Daydream's prediction for how the P3 optimization will help under different network bandwidths.

primitives. As shown in Figure 9, adding synchronizations improve the NCCL primitives by 22.8% on average when compared to the baseline.

We also verify the impact to the overall iteration time when adding synchronizations before NCCL primitives. We run the experiments on all the configurations shown in Figure 8. We find that this simple approach does not lead to performance degradation in any configuration. Instead, it could bring an improvement of up to 22%.

6.6 Priority-Based Parameter Propagation

We evaluate Daydream's prediction accuracy of applying Priority-Based Parameter Propagation (P3) to VGG-19 and ResNet-50. To reproduce the performance speedups of P3, we use a cluster of four machines with one P4000 GPU per machine (which is consistent with the evaluation setup of the P3 paper [30]). We use MXNet v1.1, and have one worker process and one parameter server process on each machine.

Figure 10 shows the iteration time of the baseline, ground truth, and prediction using Daydream under different bandwidths. Our prediction faithfully reflects the trend of P3 speedups when the network bandwidth increases. The prediction error is at most 16.2% among all the configurations we tested, and lower in most of the configurations.

We overestimate the speedup of P3, especially when training VGG-19 with a 15 or 20 Gbps network bandwidth. The reason is similar to our previous insight about NCCL primitives: when bandwidth is higher, a communication task is increasingly bottlenecked by non-network resources. In the case of MXNet, this overhead could be caused by the server processes, or the control flow of the worker processes.

7 Discussion

In this section, we discuss the adaptability, potential extensions, and some limitations of Daydream.

Why Not Simply Run the Optimizations? The main problem many ML developers face is that not all optimizations are readily available on all platforms. In fact, we are only able to evaluate the prediction accuracy of optimizations with the implementations already available (see Table 1); for the remaining ones, we highlight the flexibility of Daydream by showing that they can be represented succinctly. Most newly proposed optimizations do not have open-source implementations on **all** DNN frameworks available right away; it would be unreasonable to expect researchers to open-source their implementations and port their optimizations on all platforms. Therefore, analyzing if these optimizations can help in a deployment setting, using Daydream, can still precede the programming effort to port the optimizations. Furthermore, Daydream's profiling can be performed just once, and using that profile on a given platform, one can answer questions for many different optimizations.

Adaptability of Daydream Daydream requires support from hardware profilers. The current implementation of Daydream utilizes GPU-based profilers, and it relies on CUPTI to provide: (i) CPU and GPU traces and (ii) information about which CPU call triggered the launch of a specific GPU kernel. Adapting our design to other architectures (e.g., TPUs), would require hardware vendor profilers to provide similar traces for this new hardware.

Daydream can be also easily adapted to other ML frameworks (e.g., MXNet and TensorFlow). We built Daydream based on PyTorch, and then post-process the dumped traces to make predictions. The post-processing scripts are framework-independent. To add framework instrumentation, we need to: (i) add CUPTI (or similar tool) support, (ii) insert per-layer timestamps, and (iii) gather the gradient-to-bucket mappings for injecting the communication primitives to the dependency graph (required for PyTorch). Such instrumentation is relatively light-weight and can be easily adapted to other mainstream frameworks such as TensorFlow [3] and MXNet [12].

Training Accuracy Prediction In addition to improving iteration time, some optimizations may also affect training accuracy (e.g., AMP [42], DGC [40]); predicting the impact of optimizations on accuracy is currently outside of Daydream's scope. We leave this interesting and challenging problem for future work.

Kernel Runtime Prediction Estimating the effect of optimizations that alter existing GPU kernels or introduce new ones requires predicting the runtime of new/changed GPU kernels. When estimating performance of AMP, our estimation of kernels that use half-precision kernels was based on findings/observations from NVIDIA [42]. This generalization above for all kernels (in contrast to identifying how each kernel in isolation is affected by AMP), still leads to the low prediction errors we observe in Figure 5.

However, optimizations such as DGC [40], Reconstructing Batchnorm [35], and Gist [29] introduce newly-implemented kernels to the runtime. Accurately predicting runtime for new kernels is a challenging problem. Daydream estimates the overall runtime based on existing kernel implementations, or using guidelines from studies that highlight quantitative improvements for the proposed kernels. But if the estimated runtimes for such new kernels are inaccurate, it may lead to relatively high prediction error (Section 6.4). How much a kernel's runtime estimation error contributes to the overall prediction error depends on the training workload itself. Due to this limitation, it is hard for Daydream to accurately model algorithmic innovations (e.g., BPPSA [77] or 2nd Order Optimizations [68]), because these innovations use new GPU kernels at a massive scale, making the performance estimation with Daydream less accurate. Estimating new GPU kernels runtime is beyond the current scope of Daydream.

While Daydream cannot predict individual kernel runtime, it provides a high-level structure for kernel developers to estimate the overall performance. Developers can profile their individual kernels, and then input the profiling results into Daydream to accurately estimate the overall runtime. This approach saves the engineering effort of porting the kernel implementation into the DNN frameworks.

Concurrent Kernels Existing GPU profilers such as CUPTI usually serialize GPU kernel execution, removing all concurrency, making our performance estimation somewhat conservative. Despite this, we observe that the runtime for models with concurrent execution (e.g., GNMT) can still be predicted with high accuracy (§ 6.2). This is because the majority of computation time goes to fully connected layers (including embedding layers), which have no concurrent kernels executed in parallel with them. We leave a complete solution for concurrent kernels, requiring better support from profiling tools, as a part of future work.

8 Related Work

To help programmers understand the performance of the hardware accelerators and develop highly efficient applications, hardware vendors provide profiling tools (e.g., NVProf [48], Nsight [47], and vTune [66]) that can reveal low-level performance counters (e.g., cache hit rate, memory speed or clock rate). These tools are usually designed with general applications in mind, and expose hundreds of low-level performance counters. The fundamental limitation of all these tools is that they do not utilize application-specific knowledge.

The new generation of profiling tools feature the *application-aware* property, enabling them to deliver domain-specific (e.g., ML-specific) insights about performance to programmers. The Cloud TPU Tool [21] is an example of such a profiling tool. It correlates low-level TPU metrics with the DNN structure, and shows the performance for each DNN layer. Similarly, MXNet [12] and PyTorch [61] also have their own built-in profiling tools. These domain-specific tools can highlight performance hotspots, but are less efficient in finding optimization opportunities. In contrast, Daydream is not only *application-aware*, but also *optimization-aware*, enabling Daydream to quantitatively estimate the efficacy of different optimizations without fully implementing them.

Prior works have tried to explore what-if questions in other contexts by using low-level traces. Curtsinger *et al.* proposed a causal profiler (COZ [17]) to identify potentially unknown optimization opportunities by running performance simulation with certain functions being virtually speed-up. Unlike Daydream, COZ does not require dependencies among functions because it does not consider the cases where functions can be added or deleted (which is the case for many ML optimizations). Pourghassemi *et al.* uses the idea of COZ to analyze the performance for web browser applications [63]. For data analytic frameworks, such as Spark [82], Ousterhout *et al.* use dependency analysis to understand the overhead caused by I/O, network, and stragglers [59,60]. Daydream is designed to address a more diversified set of what-if questions, and hence requires more powerful modeling.

Prior works address what-if questions of the form "What if we can speedup task T by N times (or infinity)?", but they do not study whether existing optimizations can deliver this speedup. In the ML context, given an optimization, accurately predicting the performance of individual tasks in the dependency graph, is still an open problem. It requires additional knowledge about the kernel implementation and the architecture design. Currently Daydream can not automatically estimate the runtime of new GPU kernels. However, as we show in Section 6, even with rough estimates of per-kernel duration based on domain knowledge and reasonable assumptions, we can still achieve high overall prediction accuracy.

9 Conclusion

The efficacy of DNN optimizations can vary largely across different DNN models and deployments. Daydream is a new profiler to effectively explore the efficacy of a diverse set of DNN optimizations. Daydream achieves this goal by using three key ideas: (i) constructing a kernel-level dependency graph by utilizing vendor-provided profiling tools, while tracking dependencies among concurrently executing tasks; (ii) mapping low-level traces to DNN layers in a synchronization-free manner; (iii) introducing a set of rules for programmers to effectively describe and model different optimizations. Our evaluation shows that using Daydream, we can effectively model (i.e. predict runtime) the most common DNN optimizations, and accurately identify both optimizations that result in significant performance improvements as well as those that provide limited benefits or even slowdowns.

Acknowledgement

Daydream is a part of Project Fiddle at Microsoft Research (MSR). We thank the MSR Lab LT, especially Ricardo Bianchini and Donald Kossmann, for their enthusiastic and unwavering support of Project Fiddle. We also thank our shepherd, Swaminathan Sundararaman, the anonymous ATC reviewers, Jorgen Thelin, Shivaram Venkataraman, Deepak Narayanan, and the EcoSystem group members, especially James Gleeson, Geoffrey Yu, and Xiaodan (Serina) Tan for their constructive feedback and comments. This work was also supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, and Huawei grants.

References

[1] Eigen: A C++ linear algebra library. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[2] PyTorch Documentation. https://pytorch.org/docs/stable/index.html, 2019.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] ACL. Shared Task: Machine Translation of News. http://www.statmt.org/wmt16/translation-task.html, 2016.

[5] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[6] Jasmin Ajanovic. PCI Express*(PCIe*) 3.0 Accelerator Features. *Intel Corporation*, 10, 2008.

[7] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[8] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[9] AMD. AMD EPYC^{TM} 7601. https://www.amd.com/en/products/cpu/amd-epyc-7601, 2019.

[10] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. In *ACM SIGPLAN Notices*, volume 50, pages 173–182. ACM, 2015.

[11] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, pages 1–15, 2018.

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[16] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1–1, 2019.

[17] Charlie Curtsinger and Emery D Berger. C oz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[18] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] Google. Cloud TPU Tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools, 2018.

[22] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[23] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[24] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR 2016)*, 2016.

[25] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[26] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[28] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[29] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceeding of the 45st Annual International Symposium on Computer Architecture*, ISCA 2018, pages 776–789, 2018.

[30] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *Proceedings of Machine Learning and Systems 2019*, pages 132–145. 2019.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[32] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62. ACM, 2019.

[33] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN computation with relaxed graph substitutions. In *Proc. Conference on Systems and Machine Learning, SysML*, volume 19, 2019.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA 2017, pages 1–12, New York, NY, USA, 2017. ACM.

[35] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. Restructuring batch normalization to accelerate CNN training. *arXiv preprint arXiv:1807.01702*, 2018.

[36] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 43. ACM, 2019.

[37] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[40] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[41] Qu Lu, Wantao Liu, Jizhong Han, and Jinrong Guo. Multi-stage Gradient Compression: Overcoming the Communication Bottleneck in Distributed Deep Learning. In *International Conference on Neural Information Processing*, pages 107–119. Springer, 2018.

[42] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[43] Barton P Miller and Cui-Qing Yang. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. In *ICDCS*, pages 482–489, 1987.

[44] MLPerf. MLPerf Training Results v0.6. https://mlperf.org/training-results-0-6, 2019.

[45] NVIDIA. CUDA implementation of the standard basic linear algebra subroutines (BLAS). http://docs.nvidia.com/cuda/cublas/index.html.

[46] NVIDIA. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[47] NVIDIA. NVIDIA Nsight. https://developer.nvidia.com/tools-overview.

[48] NVIDIA. NVIDIA Profiler. docs.nvidia.com/cuda/profiler-users-guide/index.html.

[49] NVIDIA. The CUDA Profiling Tools Interface (CUPTI). https://docs.nvidia.com/cuda/cupti/index.html.

[50] NVIDIA. cudnn library developer guide v6.0. 2017.

[51] NVIDIA. A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex, 2018.

[52] NVIDIA. API Documentation of NVidia's Apex optimizers. https://nvidia.github.io/apex/optimizers.html, 2018.

[53] NVIDIA. Cuda toolkit documentation v10.0. https://docs.nvidia.com/cuda/, 2018.

[54] NVIDIA. cudnn library developer guide v 7.4.1. 2018.

[55] NVIDIA. GEFORCE® RTX 2080 Ti. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti, 2018.

[56] NVIDIA. NVIDIA Turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[57] NVIDIA. Performance reported by NCCL tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md, 2018.

[58] NVIDIA. Training With Mixed Precision: Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html, 2019.

[59] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200. ACM, 2017.

[60] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[62] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29. ACM, 2019.

[63] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):27, 2019.

[64] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[65] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[66] James Reinders. VTune performance analyzer essentials. *Intel Press*, 2005.

[67] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 18:1–18:13, Piscataway, NJ, USA, 2016. IEEE Press.

[68] Tomer Koren Kevin Regan Yoram Singer Rohan Anil, Vineet Gupta. Second Order Optimization Made Practical. *arXiv preprint arXiv:2002.09018*, 2020.

[69] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597*, 2019.

[70] Amazon Web Services. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia.

[71] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[72] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[74] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 185–196. ACM, 2008.

[75] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi^{TM}*, pages 167–188. Springer, 2014.

[76] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. *arXiv preprint arXiv:1810.08313*, 2018.

[77] Shang Wang, Yifan Bai, and Gennady Pekhimenko. Bppsa: Scaling back-propagation by parallel scan algorithm. In *Proceedings of Machine Learning and Systems 2020*, pages 451–469. 2020.

[78] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[79] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[80] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 44. ACM, 2019.

[81] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[82] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[83] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[84] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100. IEEE, 2018.

[85] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. *arXiv preprint arXiv:2006.03318*, 2020.

ALERT: Accurate Learning for Energy and Timeliness

Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, Shan Lu
The University of Chicago

Abstract

An increasing number of software applications incorporate runtime Deep Neural Networks (DNNs) to process sensor data and return inference results to humans. Effective deployment of DNNs in these interactive scenarios requires meeting latency and accuracy constraints while minimizing energy, a problem exacerbated by common system dynamics.

Prior approaches handle dynamics through either (1) system-oblivious DNN adaptation, which adjusts DNN latency/accuracy tradeoffs, or (2) application-oblivious system adaptation, which adjusts resources to change latency/energy tradeoffs. In contrast, this paper improves on the state-of-the-art by coordinating application- and system-level adaptation. ALERT, our runtime scheduler, uses a probabilistic model to detect environmental volatility and then simultaneously select both a DNN and a system resource configuration to meet latency, accuracy, and energy constraints. We evaluate ALERT on CPU and GPU platforms for image and speech tasks in dynamic environments. ALERT's holistic approach achieves more than 13% energy reduction, and 27% error reduction over prior approaches that adapt solely at the application or system level. Furthermore, ALERT incurs only 3% more energy consumption and 2% higher DNN-inference error than an oracle scheme with perfect application and system knowledge.

1 Introduction

1.1 Motivation

Deep neural networks (DNNs) have become a key workload for many computing systems due to their high inference accuracy. This accuracy, however, comes at a cost of long latency, high energy usage, or both. Successful DNN deployment requires meeting a variety of user-defined, application-specific goals for latency, accuracy, and often energy in unpredictable, dynamic environments.

Latency constraints naturally arise with DNN deployments when inference interacts with the real world as a consumer—processing data streamed from a sensor—or a producer—returning a series of answers to a human. For example, in motion tracking, a frame must be processed at camera speed [40]; in simultaneous interpretation, translation must be provided every 2–4 seconds [56]. Violating these deadlines may lead to severe consequences: if a self-driving vehicle cannot act within a small time budget, life threatening accidents could follow [53].

Accuracy and energy requirements are also common and may vary for different applications in different operating environments. On one hand, low inference accuracy can lead to software failures [67, 80]. On the other hand, it is beneficial to minimize DNN energy or resource usage to extend mobile-battery time or reduce server-operation cost [41].

These requirements are also highly dynamic. For example, the latency requirement for a job could vary dynamically depending on how much time has already been consumed by related jobs before it [53]; the power budget and the accuracy requirement for a job may switch among different settings depending on what type of events are currently sensed [1]. Additionally, the latency requirement may change based on the computing system's current context; e.g., in robotic vision systems the latency requirement can change based on the robot's latency and distance from perceived pedestrians [18].

Satisfying all these requirements in a dynamic computing environment where the inference job may compete for resources against unpredictable, co-located jobs is challenging. Although prior work addresses these problems at either the application level or system level separately, each approach by itself lacks critical information that could be used to produce better results.

At the application level, different DNN designs—with different depths, widths, and numeric precisions—provide various latency-accuracy trade-offs for the same inference task [26, 39, 42, 77, 85]. Even more dynamic schemes have been proposed that adapt the DNN by dynamically changing its structure at the beginning of [22, 61, 84, 89] or during [34, 35, 49, 52, 82, 86, 88] every inference tasks.

Although helpful, these techniques are sub-optimal without considering system-level adaptation options. For example, under energy pressure, these application-level adaptation techniques have to switch to lower-accuracy DNNs, sacrificing accuracy for energy saving, even if the energy goal could have been achieved by lowering the system power setting (if there is sufficient latency budget).

At the system level, machine learning [4, 14, 15, 51, 63, 68, 69, 79] and control theory [32, 37, 44, 45, 62, 70, 74, 93] based techniques have been proposed to dynamically assign system resources to better satisfy system and application constraints.

Unfortunately, without considering the option of application adaptions, these techniques also reach sub-optimal solutions. For example, when the current DNN offers much higher accuracy than necessary, switching to a lower-precision DNN may offer much more energy saving than any system-level adaptation techniques. This problem is exacerbated because, in the DNN design space, very small drops in accuracy enable dramatic reductions in latency, and therefore system resource requirements.

A cross-stack solution would enable DNN applications to meet multiple, dynamic constraints. However, offering such a holistic solution is non-trivial. The combination of DNN and system-resource adaptation creates a huge configuration space, making it difficult to dynamically and efficiently predict which combination of DNN and system settings will meet all the requirements optimally. Furthermore, without careful coordination, adaptations at the application and system level may conflict and cause constraint violations, like missing a latency deadline due to switching to higher-accuracy DNN and lower power setting at the same time.

1.2 Contributions

This paper presents ALERT, a cross-stack runtime system for DNN inference to meet user goals by simultaneously adapting both DNN models and system-resource settings.

Understanding the challenges We profile DNN inference across applications, inputs, hardware, and resource contention confirming there is a high variation in inference time. This leads to challenges in meeting not only latency but also energy and accuracy requirements. Furthermore, our profiling of 42 existing DNNs for image classification confirms that different designs offer a wide spectrum of latency, energy, and accuracy tradeoffs. In general, higher accuracy comes at the cost of longer latency and/or higher energy consumption. These tradeoffs offered provide both opportunities and challenges to holistic inference management (Section 2).

Run-time inference management We design ALERT, a DNN inference management system that dynamically selects and adapts a DNN and a system-resource setting together to handle changing system environments and meet dynamic energy, latency, and accuracy requirements[1] (Section 3).

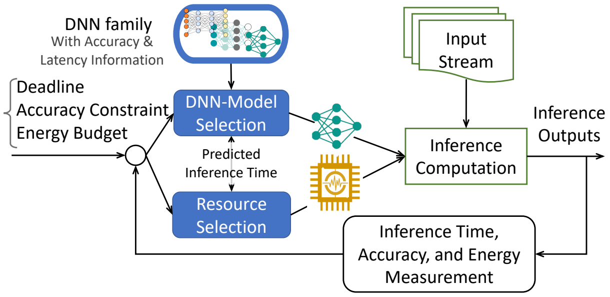


Figure 1: ALERT inference system

ALERT is a feedback-based run-time. It measures inference accuracy, latency, and energy consumption; it checks whether the requirements on these goals are met; and, it then outputs both system and application-level configurations adjusted to the current requirements and operating conditions. ALERT focuses on meeting constraints in *any* two dimensions while optimizing the third; e.g., minimizing energy given accuracy and latency requirements or maximizing accuracy given latency and energy budgets.

The key is estimating how DNN and system configurations interact to affect the goals. To do so, ALERT addresses three primary challenges: (1) the combined DNN and system configuration space is huge, (2) the environment may change dynamically (including input, available resources, and even the required constraints), and (3) the predictions must be low overhead to have negligible impact on the inference itself.

ALERT addresses these challenges with a *global slow-down factor*, a random variable relating the current runtime environment to a nominal profiling environment. After each inference task, ALERT estimates the global slow-down factor using a Kalman filter. The global slow-down factor's mean represents the expected change compared to the profile, while the variance represents the current volatility. The mean provides a single scalar that modifies the predicted latency/accuracy/energy for *every* DNN/system configuration—a simple mechanism that leverages commonality among DNN architectures to allow prediction for even rarely used configurations (tackle challenge-1), while incorporating variance into predictions naturally makes ALERT conservative in volatile environments and aggressive in quiescent ones (tackle challenge-2). The global slow-down factor and Kalman filter are efficient to implement and low-overhead (tackle challenge-3). Thus, ALERT combines the global slow-down factor with latency, power, and accuracy measurements to select the DNN and system configuration with the highest likelihood of meeting the constraints optimally.

We evaluate ALERT using various DNNs and application domains on different (CPU and GPU) machines under various constraints. Our evaluation shows that ALERT overcomes dynamic variability efficiently. Across various experimental

[1] ALERT provides probabilistic, not hard guarantees, as the latter requires much more conservative configurations, often hurting both energy and accuracy. Section 3.6 discusses this issue further.

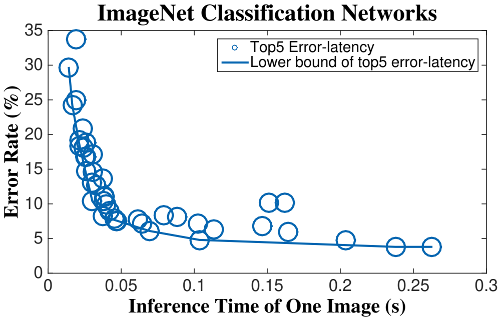


Figure 2: Tradeoffs for 42 DNNs (CPU2).

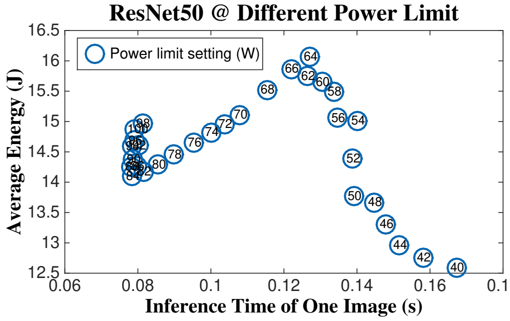


Figure 3: Tradeoffs for ResNet50 at different power settings (CPU2). (Numbers inside circles are power limit settings.)

settings, ALERT meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization. Compared to approaches that adapt at application-level or system-level only ALERT achieves more than 13% energy reduction, and 27% error reduction (Section 5).

2 Understanding Deployment Challenges

We conduct an empirical study to examine the large trade-off space offered by different DNN designs and system settings (Sec. 2.1), and the timing variability of inference (Sec. 2.2).

	Embedded	CPU1	CPU2	GPU
CPU	ARM Cortex A-15 @2.0 GHz	Core-i7 @2.2 GHz	Xeon(R) Gold 6126 @2.60GHz	Core-i7 @2.2 GHz
GPU	none	none	none	RTX 2080
Memory	DDR3 2G	DDR4 16G	DDR4 16G*12	DDR4 16G
LLC	2MB	9MB	19.25MB	9MB

Table 1: Hardware platforms used in our experiments

ID	Task	DNN Models	Datasets
IMG1	Image Classification	VGG16 [78]	ILSVRC2012 (ImageNet)
IMG2		ResNet50 [29]	
NLP1	Sentence Prediction	RNN	Penn Treebank [59]
NLP2	Question Answering	Bert [17]	Stanford Q&A Dataset (SQuAD) [71]

Table 2: ML tasks and benchmark datasets in our experiments

We use two canonical machine learning tasks, with state-of-the-art networks and common data-sets (see Table 2) on a diverse set of hardware platforms, representing embedded systems, laptops (CPU1), CPU servers (CPU2), and GPU platforms (see Table 1). The two tasks, image classification and natural language processing (NLP), are often deployed with deadlines—e.g., for motion tracking [40] and simultaneous interpretation [56]—and both have received wide attention leading to a diverse set of DNN models.

2.1 Understanding the Tradeoffs

Tradeoffs from DNNs We run **all** 42 image classification models provided by the Tensorflow website [76] on the 50000 images from ImageNet [16], and measure their average latency, accuracy (error rate), and energy consumption. The results from CPU2 are shown in Figure 2. We can clearly see two trends from the figure, which hold on other machines.

First, different DNN models offer a *wide* spectrum of accuracy (error rate in figure), latency, and energy. As shown in the figure, the fastest model runs almost 18× faster than the slowest one and the most accurate model has about 7.8× lower error rate than the least accurate. These models also consume a wide range—more than 20×—of energy usage.

Second, there is no magic DNN that offers both the best accuracy and the lowest latency, confirming the intuition that there exists a tradeoff between DNN accuracy and resource usage. Of course, some DNNs offer better tradeoffs than others. In Figure 2, all the networks sitting above the lower-convex-hull curve represent sub-optimal tradeoffs.

Tradeoffs from system settings We run ResNet50 under 31 power settings from 40–100W on CPU2. We consider a sensor processing scenario with periodic inputs, setting the period to the latency under 40W cap. We then plot the average energy consumed for the whole period (run-time plus idle energy) and the average inference latency in Figure 3.

The results reflect two trends, which hold on other machines. First, a large latency/energy space is available by changing system settings. The fastest setting (100W) is more than 2× faster than the slowest setting (40W). The most energy-hungry setting (64W) uses 1.3× more energy than the least (40W). Second, there is no easy way to choose the best setting. For example, 40W offers the lowest energy, but highest latency. Furthermore, most of these points are sub-optimal in terms of energy and latency tradeoffs. For example, 84W should be chosen for extremely low latency deadlines, but all other nearby points (from 52–100) will harm latency, energy or both. Additionally, when deadlines change or when there is resource contention, the energy-latency curve also changes and different points become optimal.

Summary: DNN models and system-resource settings offer a huge trade-off space. The energy/latency tradeoff space is not smooth (when accounting for deadlines and idle power) and optimal operating points cannot be found with simple gradient-based heuristics. Thus, there is a great

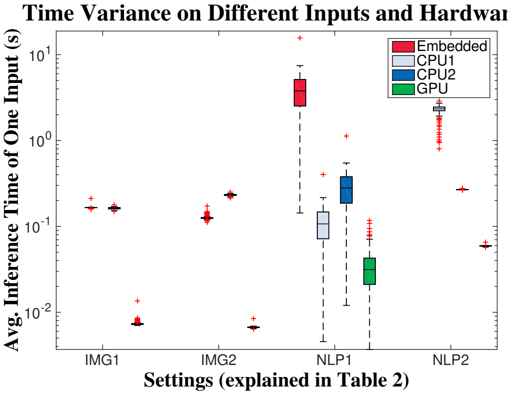


Figure 4: Latency variance across inputs for different tasks and hardware (Most tasks have 3 boxplots for 3 hardware platforms, CPU1-2, GPU from left to right; NLP1 has an extra boxplot for Embedded; other tasks run out of memory on Embedded; every box shows the 25th–75th percentile; points beyond the whiskers are >90th or <10th).

opportunity and also a great challenge in picking different DNN models and system-resource settings to satisfy inference latency, accuracy, and energy requirements.

2.2 Understanding Variability

To understand how DNN-inference varies across inputs, platforms, and run-time environment and hence how (not) helpful is off-line profiling, we run a set of experiments below, where we feed the network one input at a time and use 1/10 of the total data for warm up, to emulate real-world scenarios. We plot the inference latency without and with co-located jobs in Figure 4 and 5, and we see several trends.

First, deadline violation is a realistic concern. Image classification on video has deadlines ranging from 1 second to the camera latency (e.g., 1/60 seconds) [40]; the two NLP tasks, have deadlines around 1 second [64]. There is clearly no single inference task that meets all deadlines on all hardware.

Second, the inference variation among inputs is relatively small particularly when there are no co-located jobs (Fig. 4), except for that in NLP1, where this large variance is mainly caused by different input lengths. For other tasks, outlier inputs exist but are rare.

Third, the latency and its variation across inputs are both greatly affected by resource contention. Comparing Figure 5 with Figure 4, we can see that the co-located job has increased both the median latency, the tail inference, and the difference between these two for all tasks on all platforms. This trend also applies to other contention cases.

While the discussion above is about latency, similar conclusions apply to inference accuracy and energy: the accuracy typically drops to close to 0 when the inference time exceeds the latency requirement, and the energy consumption naturally changes with inference time.

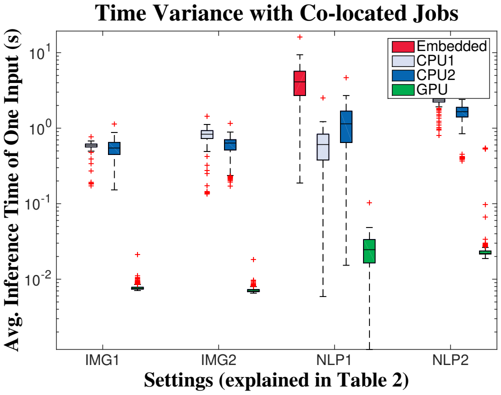


Figure 5: Latency variance with co-located jobs (the memory-intensive STREAM benchmark [60] co-located on Embedded, CPU1-2; GPU-intensive Backprop [8] co-located on GPU)

Summary: Deadline violations are realistic concerns and inference latency varies greatly across platforms, under contention, and sometimes across inputs. Clearly, sticking to one static DNN design across platforms and workloads leads to an unpleasant trade-off: always meeting the deadline by sacrificing accuracy or energy in most settings, or achieving a high accuracy some times but exceeding the deadline in others. Furthermore, it is also sub-optimal to make run-time decisions based solely on off-line profiling, considering the variation caused by run-time contention.

2.3 Understanding Potential Solutions

We now show how confining adaptation to a single layer (just application or system) is insufficient. We run the ImageNet classification on *CPU1*. We examine a range of latency (0.1s-0.7s) and accuracy constraints (85%-95%), and try meeting those constraints while minimizing energy by either (1) configuring just the DNN (selecting a DNN from a family, like that in Figure 2) or (2) configuring just the system (by selecting resources to control energy–latency tradeoffs as in Figure 3). We compare these single-layer approaches to one that simultaneously picks the DNN and system configuration. As we are concerned with the ideal case, we create oracles by running 90 inputs in all possible DNN and system configurations, from which we find the best configuration for each input. The App-level oracle uses the default system setting. The Sys-level oracle uses the default (highest accuracy) DNN.

Figure 6 shows the results. As we have a three dimensional problem—meeting accuracy and latency constraints with minimal energy—we linearize the constraints and show them on the x-axis (accuracy is faster changing, with latency slower, so each latency bin contains all accuracy goals). There are several important conclusions here. First, the App-only approach meets all possible accuracy and latency constraints, while the Sys-only approach cannot meet any constraints below 0.3s. Second, across the entire constraint range, App-

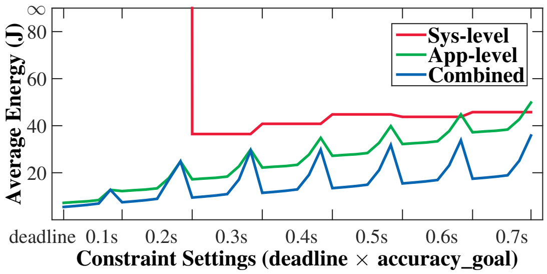


Figure 6: Minimize energy task with latency and accuracy constraint @ CPU1. (∞ means unable to meet the constraints)

only consumes significantly more energy than Combined (60% more on average). The intuition behind Combined's superiority is that there are discrete choices for DNNs; so when one is selected, there are almost always energy saving opportunities by tailoring resource usage to that DNN's needs.

Summary: Combining DNN and system level approaches achieves better outcomes. If left solely to the application, energy will be wasted. If left solely to the system, many achievable constraints will not be met.

3 ALERT Run-time Inference Management

ALERT's runtime system navigates the large tradeoff space created by *combining* DNN-level and system-level adaptation. ALERT meets user-specified latency, accuracy, and energy constraints and optimization goals while accounting for runtime variations in environment or the goals themselves.

3.1 Inputs & Outputs of ALERT

ALERT's inputs are specifications about (1) the adaption options, including a set of DNN models $\mathbb{D} = \{d_i \mid i = 1 \cdots K\}$ and a set of system-resource settings, expressed as different power-caps $\mathbb{P} = \{P_j \mid j = 1 \cdots L\}$; and (2) the user-specified requirements on latency, accuracy, and energy usage, which can take the form of meeting constraints in any two of these three dimensions while optimizing the third. ALERT's output is the DNN model $d_i \in \mathbb{D}$ and the system-resource setting $p_j \in \mathbb{P}$ for the next inference-task input.

Formally, ALERT selects a DNN d_i and a system-resource setting p_j to fulfill *either* of these user-specified goals.

Maximizing inference accuracy q (minimizing error) for an energy budget $\mathbf{E}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$:

$$\arg\max_{i,j} q_{i,j} \quad \text{s.t. } e_{i,j} \leq \mathbf{E}_{\text{goal}} \wedge t_{i,j} \leq \mathbf{T}_{\text{goal}} \tag{1}$$

Minimizing the energy use e for an accuracy goal $\mathbf{Q}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$:

$$\arg\min_{i,j} e_{i,j} \quad \text{s.t. } q_{i,j} \geq \mathbf{Q}_{\text{goal}} \wedge t_{i,j} \leq \mathbf{T}_{goal} \tag{2}$$

We omit the discussion of meeting energy and accuracy constraints while minimizing latency as it is a trivial extension of the discussed techniques and we believe it to be the least practically useful. We also omit the problem of optimizing all three dimensions, as it creates a feasibility problem, leaving nothing for optimization—lowest latency and highest accuracy are impractical to achieve simultaneously.

Generality Along the DNN-adaptation side, the input DNN set can consist of any DNNs that offer different accuracy, latency, and energy tradeoffs; e.g., those in Figure 3. In particular, ALERT can work with either or both of the broad classes of DNN adaptation approaches that have arisen recently, including: (1) traditional DNNs where the adaptation option should be selected prior to starting an inference task [20,22,61,84,89] and (2) anytime DNNs that produce a series of outputs as they execute [34,35,49,52,82,86,88]. These two classes are similar in that they both vary things like the network depth or width to create latency/accuracy tradeoffs.

On the system-resource side, ALERT uses a *power cap* as the proxy to system resource usage. Since both hardware [13] and software resource managers [33,72,90] can convert power budgets into optimal performance resource allocations, ALERT is compatible with many different schemes from both commercial products and the research literature.

3.2 ALERT Workflow

ALERT works as a feedback controller. It follows four steps to pick the DNN and resource settings for each input n:

1) Measurement. ALERT records the processing time, energy usage, and computes inference accuracy for $n-1$.

2) Goal adjustment. ALERT updates the time goal T_{goal} if necessary, considering the potential latency-requirement variation across inputs. In some inference tasks, a set of inputs share one combined requirement (e.g., in the NLP1 task in Table 2, all the words in a sentence are processed by a DNN one by one and share one sentence-wise deadline) and hence delays in previous input processing could greatly shorten the available time for the next input [1,47]. Additionally, ALERT sets the goal latency to compensate for its own, worst-case overhead so that ALERT itself will not cause violations.

3) Feedback-based estimation. ALERT computes the expected latency, accuracy, and energy consumption for every combination of DNN model and power setting.

4) Picking a configuration. ALERT feeds all the updated estimations of latency, accuracy, and energy into Eqs. 1 and 2, and gets the desired DNN model and power-cap setting for n.

The key task is step 3: the estimation needs to be accurate and fast. In the remainder of this section, we discuss key ideas and the exact algorithm of our feedback-based estimation.

3.3 Key Ideas of ALERT Estimation

Strawman Solving Eqs. 1 and 2 would be trivially easy if the deployment environment is guaranteed to match the training and profiling environment: we could estimate $t_{i,j}$ to be the average (or worst case, etc) inference time $t_{i,j}^{\text{prof}}$ over a set of profiling inputs under model d_i and power setting p_j. However, this approach does not work given the dynamic input, contention, and requirement variation.

Next, we present the key ideas behind how ALERT estimates the inference latency, accuracy, and energy consumption under model d_i and power setting p_j.

How to estimate the inference latency $t_{i,j}$? To handle the run-time variation, a potential solution is to apply an estimator, like a Kalman filter [55], to make dynamic predictions based on recent history about inferences under model d_i and power p_j. The problem is that most models and power settings will not have been picked recently and hence would have no recent history to feed into the estimator. This problem is a direct example of the challenge imposed by the large space of combined application and system options.

Idea 1: Handle the large selection space with a single scalar value. To make effective online estimation for *all* combinations of models and power settings, ALERT introduces a *global slow-down factor* ξ to capture how the current environment differs from the profiled environment (e.g., due to co-running processes, input variation, or other changes). Such an environmental slow-down factor is independent from individual model or power selection. It can fully leverage execution history, no matter which models and power settings were recently used; it can then be used to estimate $t_{i,j}$ based on $t_{i,j}^{\text{prof}}$ for all d_i and p_j combinations.

Applying a *global* slowdown factor for *all* combinations of application and system-level settings is crucial for ALERT to make quick decisions for every inference task. Although it is possible that some perturbations may lead to different slowdowns for different configurations, the slight loss of accuracy here is out-weighed by the benefit of having a simple mechanism that allows prediction even for configurations that have not been used recently.

This idea is also novel for ALERT, as previous cross-stack management systems all use much more complicated models to estimate and select different setting combinations (e.g., using model predictive control to estimate combinations of settings [57]). ALERT's global slowdown factor is based on several unique features of DNN families that accomplish the same task with different accurarcy/latency tradeoffs. We categorize these features as: (1) similarity of code paths and (2) proportionality of structure. The first is based on the observation that DNNs do not have complex conditional code dependences, so we do not need to worry about the case where different inputs would exercise very different code paths. Thus, what ALERT learns about latency, accuracy, and energy for one input will always inform it about future inputs. The second feature refers to the fact that as DNNs in a family scale in latency, the proportion of different operations tend to be similar, so what ALERT learns about one DNN in the family generally applies to other DNNs in the same family. These properties of DNNs do not hold for many other types of software, where different inputs or additional functionality can invoke entirely different code paths, with different resource requirements or responses.

How to estimate the accuracy under a deadline? Given a deadline $\mathbf{T}_{\text{goal}}$, the inference accuracy delivered by model d_i and power setting p_j is determined by three factors, as shown in Eq. 3: (1) whether the inference result, which takes time $t_{i,j}$, can be generated before the deadline $\mathbf{T}_{\text{goal}}$; (2) if yes, the accuracy is determined by the model d_i;[2] (3) if not, the accuracy drops to that offered by a backup result q_{fail}. For traditional DNN models, without any output at the deadline, a random guess will be used and q_{fail} will be much worse than q_i. For anytime DNN models that output multiple results as they are ready, the backup result is the latest output [34, 35, 49, 52, 82, 86, 88], which we discuss more in Section 3.5.

$$q_{i,j}[\mathbf{T}_{\text{goal}}] = \begin{cases} q_i & \text{, if } t_{i,j} \leq \mathbf{T}_{\text{goal}} \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \tag{3}$$

A potential solution to estimate accuracy $q_{i,j}$ at the deadline $\mathbf{T}_{\text{goal}}$ is to simply feed the estimated $t_{i,j}$ into Eq. 3. However, this simple approach fails to account for two issues. First, while DNNs are generally well-behaved, significant tail effects are possible (see Figure 4). Second, Eq. 3 is not linear, and is best understood as a step function, where a failure to complete inference by the deadline results in a worthless inference output (q_{fail}). Combined, these two issues mean that for tail inputs, inference will produce a worthless result; i.e., accuracy is not proportional to latency, but can easily fall to zero for tail inputs. The tail will, of course, be increased if there is any unexpected resource contention. Therefore, the simple approach of using the mean latency prediction fails to account for the non-linear affects of latency on accuracy.

Idea 2: handle the runtime variation and account for tail behavior To handle the run-time variability mentioned in Section 1, ALERT treats the execution time $t_{i,j}$ and the global slow-down factor ξ as *random variables* drawn from a normal distribution. ALERT uses a recently proposed extension to the Kalman filter to adaptively update the noise covariance [2]. While this extension was originally proposed to produce better estimates of the mean, a novel approach in ALERT is using this covariance estimate as a measure of system volatility. ALERT uses this Kalman filter extension to predict not just the mean accuracy, but also the likelihood of meeting the accuracy requirements in the current operating environment. Section 5.3 shows the advantages of our extensions.

[2] Since it could be infeasible to calculate the exact inference accuracy at run time, ALERT uses the average training accuracy of the selected DNN model d_i, denoted as q_i, as the inference accuracy, as long as the inference computation finishes before the specified deadline.

How to minimize energy or satisfy energy constraints? Minimizing energy or satisfying energy constraints is complicated, as the energy is related to, but cannot be easily calculated by, the complexity of the selected model d_i and the power cap p_j. As discussed in Section 2.2, the energy consumption includes both that used during the inference under a given model d_i and that used during the inference-idle period, waiting for the next input. Consequently, it is not straightforward to decide which power setting to use.

Idea 3. ALERT leverages insights from previous research, which shows that energy for latency-constrained systems can be efficiently expressed as a mathematical optimization problem [7, 48, 50, 62]. These frameworks optimize energy by scheduling available configurations in time. Time is assigned to configurations so that the average performance hits the desired latency target and the overall energy (including idle energy) is minimal. The key is that while the configuration space is large, the number of constraints is small (typically just two). Thus, the number of configurations assigned a non-zero time is also small (equal to the number of constraints) [48]. Given this structure, the optimization problem can be solved using a binary search over available configurations, or even more efficiently with a hash table [62].

The only difficulty applying prior work to ALERT is that prior work assumed there was only a single job running at a time, while ALERT assumes that other applications might contend for resources. Thus, ALERT cannot assume that there is a single system-idle state that will be used whenever the DNN is not executing. To address this challenge, ALERT continually estimates the system power when DNN inference is idle (but other non-inference tasks might be active), $p_{DNNidle}$, transforming Eq. 1 is transformed into:

$$\arg\max_{i,j} q_{i,j}[\mathbf{T}_{\text{goal}}] \quad \text{s.t. } p_{i,j}\cdot t_{i,j} + p_{DNNidle}\cdot t_{DNNidle} \leq \mathbf{E}_{\text{goal}} \tag{4}$$

3.4 ALERT Estimation Algorithm

Global Slow-down Factor ξ. As discussed in Idea-1, ALERT uses ξ to reflect how the run-time environment differs from the profiling environment. Conceptually, if the inference task under model d_i and power-cap p_j took time $t_{i,j}$ at run time and took $t_{i,j}^{\text{prof}}$ on average to finish during profiling, the corresponding ξ would be $t_{i,j}/t_{i,j}^{prof}$. ALERT estimates ξ using recent execution history under any model or power setting.

Specifically, after an input $n-1$, ALERT computes $\xi^{(n-1)}$ as the ratio of the observed time $t_{i,j}^{(n-1)}$ to the profiled time $t_{i,j}^{\text{prof}}$, and then uses a Kalman Filter[3] to estimate the mean $\mu^{(n)}$ and variance $(\sigma^{(n)})^2$ of $\xi^{(n)}$ at input n. ALERT's formulation is defined in Eq. 5, where $K^{(n)}$ is the Kalman gain variable; R is a constant reflecting the measurement noise; $Q^{(n)}$ is the process noise capped with $Q^{(0)}$. We set a forgetting factor of process variance $\alpha = 0.3$ [2]. ALERT initially sets $K^{(0)} = 0.5$, $R = 0.001$, $Q^{(0)} = 0.1$, $\mu^{(0)} = 1$, $(\sigma^{(0)})^2 = 0.1$, following the standard convention [55].

$$\begin{cases} Q^{(n)} = \max\{Q^{(0)}, \alpha Q^{(n-1)} + (1\text{-}\alpha)(K^{(n-1)}y^{(n-1)})^2\} \\ K^{(n)} = \dfrac{(1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)}}{(1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)} + R} \\ y^{(n)} = t_{i,j}^{(n-1)}/t_{i,j}^{\text{prof}} - \mu^{(n-1)} \\ \mu^{(n)} = \mu^{(n-1)} + K^{(n)}y^{(n)} \\ (\sigma^{(n)})^2 = (1-K^{(n-1)})(\sigma^{(n-1)})^2 + Q^{(n)} \end{cases} \tag{5}$$

Then, using $\xi^{(n)}$, ALERT estimates the inference time of input n under any model d_i and power cap p_j: $t_{i,j}^{(n)} = \xi^{(n)} * t_{i,j}^{\text{prof}}$.

Probability of meeting the deadline. Given the Kalman Filter estimation for the global slowdown factor, we can calculate $Pr_{i,j}$, the probability that the inference completes before the deadline T_{goal}. ALERT computes this value using a cumulative distribution function (CDF) based on the normal distribution of $\xi^{(n)}$ estimated by the Kalman Filter:

$$\begin{aligned} Pr_{i,j} &= Pr[\xi^{(n)}\cdot t_{i,j}^{\text{prof}} \leq T_{goal}] = CDF(\xi^{(n)}\cdot t_{i,j}^{\text{prof}}, T_{goal}) \\ &= CDF(\mu^{(n)}\cdot t_{i,j}^{\text{prof}}, \sigma^{(n)}, T_{goal}) \end{aligned} \tag{6}$$

Accuracy. As discussed in Idea-2, ALERT computes the estimated inference accuracy $\hat{q}_{i,j}[\mathbf{T}_{\text{goal}}]$ by considering $t_{i,j}$ as a random variable that follows normal distribution with its mean and variance computed based on that of ξ. Here $q_{i,j}$ represents the inference accuracy when the DNN inference finishes before the deadline, and q_{fail} is the accuracy of a random guess:

$$\begin{aligned} \hat{q}_{i,j}[\mathbf{T}_{goal}] &= E(q_{i,j}[\mathbf{T}_{goal}] \mid t_{i,j}^{(n)}) \\ &= E(q_{i,j}[\mathbf{T}_{goal}] \mid \xi^{(n)}\cdot t_{i,j}^{\text{prof}}) \\ &= Pr_{i,j}\cdot q_{i,j} + (1-Pr_{i,j})\cdot q_{fail} \\ \xi^{(n)} &\sim \mathcal{N}(\mu^{(n)}, (\sigma^{(n)})^2) \end{aligned} \tag{7}$$

Energy. As discussed in Idea-3, ALERT predicts energy consumption by separately estimating energy during (1) DNN execution: estimated by multiplying the power limit by the estimated latency and (2) between inference inputs: estimated based on the recent history of inference idle power using the Kalman Filter in Eq. 8. $\phi^{(n)}$ is the predicted DNN-idle power ratio, $M^{(n)}$ is process variance, S is process noise, V is measurement noise, and $W^{(n)}$ is the Kalman Filter gain. ALERT initially sets $M^{(0)} = 0.01$, $S = 0.0001$, $V = 0.001$.

$$\begin{cases} W^{(n)} = \dfrac{M^{(n-1)} + S}{M^{(n-1)} + S + V} \\ M^{(n)} = (1-W^{(n)})(M^{(n-1)} + S) \\ \phi^{(n)} = \phi^{(n-1)} + W^{(n)}(p_{\text{idle}}/p_{i,j}^{(n-1)} - \phi^{(n-1)}) \end{cases} \tag{8}$$

[3] A Kalman Filter is an optimal estimator that assumes a normal distribution and estimates a varying quantity based on multiple potentially noisy observations [55].

ALERT then predicts the energy by Eq. 9. Unlike Eq. 7 that uses probabilistic estimates, energy estimation is calculated without the notion of probability. The inference power is the same no matter the inference misses or meets the deadline, as ALERT sets power limits. Therefore it is safe to estimate the energy by its mean without considering the distribution of its possible latency. See our extended report [87] on estimating energy by its worst case latency percentile.

$$e_{i,j}^{(n)} = p_{i,j} \cdot \xi^{(n)} \cdot t_{i,j}^{\text{prof}} + \phi^{(n)} \cdot p_{i,j} \cdot (\mathbf{T}_{goal} - (\xi^{(n)} \cdot t_{i,j}^{\text{prof}})) \quad (9)$$

3.5 Integrating ALERT with Anytime DNNs

An anytime DNN is an inference model that outputs a series of increasingly accurate inference results—o_1, o_2, ... o_k, with o_t more reliable than o_{t-1}. A variety of recent works [35,49,52,82,86,88] have proposed DNNs supporting anytime inference, covering a variety of problem domains. ALERT easily works with not only traditional DNNs but also Anytime DNNs. The only change is that q_{fail} in Eq. 3 no longer corresponds to a random guess. That is, when the inference could not generate its final result o_k by the deadline $\mathbf{T}_{\text{goal}}$, an earlier result o_x can be used with a much better accuracy than that of a random guess. The updated accuracy equation is below:

$$q_{.,j} = \begin{cases} q_k & \text{, if } t_{k,j} \leq \mathbf{t}_{\text{goal}} \\ q_{k-1} & \text{, if } t_{k-1,j} \leq \mathbf{t}_{\text{goal}} < t_{k,j} \\ & \dots \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \quad (10)$$

Existing anytime DNNs consider latency but not energy constraints—an anytime DNN will keep running until the latency deadline arrives and the last output will be delivered to the user. ALERT naturally improves Anytime DNN energy efficiency, stopping the inference sometimes before the deadline based on its estimation to meet not only latency and accuracy, but also energy requirements.

Furthermore, ALERT can work with a set of traditional DNNs and an Anytime DNN together to achieve the best combined result. The reason is that Anytime DNNs generally sacrifice accuracy for flexibility. When we feed a group of traditional DNNs and one Anytime DNN to construct the candidacy set $\mathbb{D}$, with Eq. 7, ALERT naturally selects the Anytime DNN when the environment is changing rapidly (because the expected accuracy of an anytime DNN will be higher given that variance), and the regular DNN, which has slightly higher accuracy with similar computation, when it is stable, getting the best of both worlds.

In our evaluation, we will use the nested design from [86], which provides a generic coverage of anytime DNNs.

3.6 Limitations and Discussions

Assumptions of the Kalman Filter. ALERT's prediction, particularly the Kalman Filter, relies on the feedback from recent input processing. Consequently, it requires at least one input to react to sudden changes. Additionally, the Kalman filter formulations assume that the underlying distributions are normal, which may not hold in practice. If the behavior is not Gaussian, the Kalman filter will produce bad estimations for the mean of ξ for some amount of time.

ALERT is specifically designed to handle data that is not drawn from a normal distribution, using the Kalman Filter's covariance estimation to measure system volatility and accounting for that in the accuracy/energy estimations. Consequently, after just 2–3 such bad predictions of means, the estimated variance will increase, which will then trigger ALERT to pick anytime DNN over traditional DNNs or pick a low-latency traditional DNN over high-latency ones, because the former has a higher expected accuracy under high variance. So—worst case—ALERT will choose a DNN with slightly less accuracy than what could have been used with the right model. Users can also compensate for extremely aberrant latency distributions by increasing the value of $Q^{(0)}$ in Eq. 5. Section 5.3 shows ALERT performs well even when the distribution is not normal.

Probabilistic guarantees. ALERT provides probabilistic, not hard, guarantees. As ALERT estimates not just average timing, but the distributions of possible timings, it can provide arbitrarily many nines of assurance that it will meet latency or accuracy goals but cannot provide 100% guarantee (see our extended report [87] on how to configure ALERT to provide guarantees with a specific probability). Providing 100% guarantees requires the worst case execution time (WCET), an upper bound on the highest possible latency. ALERT does not assume the availability of such information and hence cannot provide hard guarantees [6].

Safety guarantees. While ALERT does not explicitly model safety requirements, it can be configured to prioritize accuracy over other dimensions. When users particularly value safety (e.g., auto-driving), they could set a high accuracy requirement or even remove the energy constraints.

Concurrent inference jobs. ALERT is currently designed to support one inference job at a time. To support multiple concurrent inference jobs, future work needs to extend ALERT to coordinate across these concurrent jobs. We expect the main idea of ALERT, such as using a global slowdown factor to estimate system variation, to still apply.

Finally, how the inference behaves ultimately depends not only on ALERT, but also on the DNN models and system-resource setting options. As shown in Section 5, ALERT helps make the best use of supplied DNN models, but does not eliminate the difference between different DNN models.

4 Implementation

We implement ALERT for both CPUs and GPUs. On CPUs, ALERT adjusts power through Intel's RAPL interface [13], which allows software to set a hardware power limit. On

	Run-time environment setting		
Default	Inference task has no co-running process		
Memory	Co-locate with memory-hungry STREAM [60] (@CPU) Co-locate with Backprop from Rodinia-3.1 [8] (@GPU)		
Compute	Co-locate with Bodytrack from PARSEC-3.0 [5] (@CPU) Co-locate with the forward pass of Backprop [8] (@GPU)		
	Ranges of constraint setting		
Latency	0.4x–2x mean latency* of the largest Anytime DNN		
Accuracy	Whole range achievable by trad. and Anytime DNN		
Energy	Whole feasible power-cap ranges on the machine		
Task	Trad. DNN	Anytime [86]	Fixed deadline?
Image Classifi.	Sparse ResNet	Depth-Nest	Yes
Sentence Pred.	RNN	Width-Nest	No
Scheme ID	DNN selection		Power selection
Oracle	Dynamic optimal		Dynamic optimal
$\text{Oracle}_{\text{Static}}$	Static optimal		Static optimal
App-only	One Anytime DNN		System Default
Sys-only	Fastest traditional DNN		State-of-Art [37]
No-coord	Anytime DNN w/o coord. with Power		State-of-Art [37]
ALERT	ALERT default		ALERT default
$\text{ALERT}_{\text{Any}}$	ALERT w/o traditional DNNs		ALERT default
$\text{ALERT}_{\text{Trad}}$	ALERT w/o Anytime DNNs		ALERT default

Table 3: Settings and schemes under evaluation (* measured under default setting without resource contention)

GPUs, ALERT uses PyNVML to control frequency and builds a power-frequency lookup table. ALERT can also be applied to other approaches that translate power limits into settings for combinations of resources [33, 36, 72, 90].

In our experiments, ALERT considers a series of power settings within the feasible range with 2.5W interval on our test laptop and a 5W interval on our test CPU server and GPU platform, as the latter has a wider power range than the former. The number of power buckets is configurable.

ALERT incurs small overhead in both scheduler computation and switching from one DNN/power-setting to another, just 0.6–1.7% of an input inference time. We explicitly account for overhead by subtracting it from the user-specified goal (see step 2 in Section 3.2).

Users may set goals that are not achievable. If ALERT cannot meet all constraints, it prioritizes latency highest, then accuracy, then power. This hierarchy is configurable.

5 Experimental Evaluation

We apply ALERT to different inference tasks on both CPU and GPU with and without resource contention from co-located jobs. We set ALERT to (1) reduce energy while satisfying latency and accuracy requirements and (2) reduce error rates while satisfying latency and energy requirements. We compare ALERT with both oracle and state-of-the-art schemes and evaluate detailed design decisions.

5.1 Methodology

Experimental setup. We use the three platforms listed in Table 1: *CPU1*, *CPU2*, and *GPU*. On each, we run inference tasks[4], image classification and sentence prediction, under three different resource-contention scenarios:

- No contention: the inference task is the only job running, referred to as "Default";
- Memory dynamic: the inference task runs together with a memory-intensive job that repeatedly stops and restarts, representing dynamic memory resource contention, referred to as "Memory";
- Computation dynamic: the inference task runs together with a computation-intensive job that repeatedly stops and restarts, representing dynamic computation resource contention, referred to as "Compute".

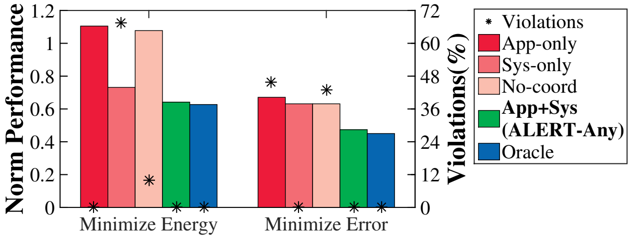


Figure 7: Average performance normalized to $\text{Oracle}_{\text{Static}}$. Violations% is %-of-constraint-settings under which a scheme incurs >10% violation of all inputs. (Smaller is better)

Schemes in evaluation. We give ALERT three different DNN sets, traditional DNN models ($\text{ALERT}_{\text{Trad}}$), an Anytime DNN ($\text{ALERT}_{\text{Any}}$), and both (ALERT), and compare it with two oracle and three state-of-the-art schemes (Table 3).

The two *Oracle*$_*$ schemes have perfect predictions for every input under every DNN/power setting (i.e., impractical). Specifically, the "Oracle" allows DNN/power settings to change across inputs, representing the best possible results; the "$\text{Oracle}_{\text{Static}}$" has one fixed setting across inputs, representing the best results without dynamic adaptation.

The three state-of-the-art approaches include the following:

- "App-only" conducts adaptation only at the application level through an Anytime DNN [86];
- "Sys-only"adapts only at the system level following an existing resource-management system that minimizes energy under soft real-time constraints [62][5] and uses the fastest candidate DNN to avoid latency violations;
- "No-coord" uses *both* the Anytime DNN for application adaptation *and* the power-management scheme [62] to adapt power, but with these two working independently.

5.2 Overall Results

Table 4 shows the results for all schemes for different tasks on different platforms and environments. Each cell shows

[4] For GPU, we only run image classification task there, as the RNN-based sentence prediction task is better suited for CPU [91].

[5] Specifically, this adaptation uses a feedback scheduler that predicts inference latency based on Kalman Filter.

Plat.	DNN	Work.	ALERT	ALERT-Any	Sys-only	App-only	No-coord	Oracle	ALERT	ALERT-Any	Sys-only	App-only	No-coord	Oracle
			Energy in Minimizing Energy Task						Error Rate in Minimizing Error Task					
CPU1	Sparse Resnet	Idle	0.64	0.68	1.08^{19}	1.19	0.94^{1}	0.64	0.91	0.92	1.35	1.02^{3}	0.91^{3}	0.89
		Comp.	0.57	0.58	0.80^{19}	1.30	1.39^{1}	0.57	0.38	0.39	0.51	1.35^{24}	0.39^{6}	0.36
		Mem.	0.53	0.55	0.76^{19}	1.43	1.37^{2}	0.53	0.34	0.34	0.46	1.47^{28}	0.39^{2}	0.33
	RNN	Idle	0.61	0.65	1.01^{30}	1.34	0.95^{2}	0.61	0.87	0.87	0.87	0.87^{21}	0.87^{14}	0.86
		Comp.	0.60	0.57	0.93^{30}	1.21	1.26^{5}	0.60	0.42	0.44	0.50	0.46^{28}	0.46^{23}	0.42
		Mem.	0.54	0.56	0.95^{31}	1.45	1.24^{9}	0.54	0.45	0.45	0.50	0.57^{28}	0.54^{27}	0.44
CPU2	Sparse Resnet	Idle	0.93	0.88	0.96^{20}	0.99	1.18	0.91	0.68	0.68	0.97	0.79^{2}	0.71^{24}	0.66
		Comp.	0.59	0.57	0.60^{23}	1.00	1.01	0.58	0.58	0.57	0.85	0.74^{16}	0.71^{29}	0.55
		Mem.	0.38	0.37	0.39^{19}	0.65	0.63^{13}	0.38	0.24	0.82	0.32	0.33^{17}	0.75^{31}	0.21
	RNN	Idle	0.87	0.99	0.80^{34}	1.04	1.00^{6}	0.83	0.84	0.85	0.99	0.89^{14}	0.89^{1}	0.84
		Comp.	0.60	0.60	0.55^{34}	0.99	0.86^{7}	0.60	0.51	0.52	0.60	0.53^{21}	0.54^{17}	0.52
		Mem.	0.52	0.51	0.43^{33}	0.70	0.85^{14}	0.52	0.26	0.27	0.31	0.28^{21}	0.27^{17}	0.26
GPU	Sparse Resnet	Idle	0.97	0.99	0.92^{20}	1.36	1.37	0.92	0.90	0.92	1.22	1.09^{2}	1.74^{12}	0.86
		Comp.	0.96	0.97	0.94^{20}	1.66	1.77	0.89	0.32	0.34	1.28	1.21^{23}	2.50^{18}	0.30
		Mem.	0.97	1.01	0.91^{20}	1.39	1.43	0.91	0.89	0.92	1.22	1.11^{2}	1.81^{14}	0.86
Harmonic mean			0.64	0.64	0.73^{27}	1.11	1.08^{4}	0.62	0.46	0.47	0.63	0.67^{16}	0.63^{15}	0.45

Table 4: Average energy consumption and error rate normalized to *Oracle*$_{\text{Static}}$, smaller is better. (Each cell is averaged over 35–40 constraint settings; superscript: # of constraint settings violated for >10% inputs and hence excluded from energy average.)

the average energy or accuracy under 35–40 combinations of latency, accuracy, and energy constraints (the settings are detailed in Table 3), normalized to the Oracle$_{\text{Static}}$ result. Figure 7 compares these results, where lower bars represent better results and lower *s represent fewer constraint violations. ALERT and ALERT $_{\text{Any}}$ both work very well for all settings. They outperform state-of-the-art approaches, which have a significant number of constraint violations, as visualized by the many superscripts in Table 4 and the high * positions in Figure 7. ALERT outperforms Oracle$_{\text{Static}}$ because it adapts to dynamic variations. ALERT also comes very close to the theoretically optimal Oracle.

Comparing with Oracles. As shown in Table 4, ALERT achieves 93-99% of Oracle's energy and accuracy optimization while satisfying constraints. Oracle$_{\text{static}}$, the baseline in Table 4, represents the best one can achieve by selecting 1 DNN model and 1 power setting for all inputs. ALERT greatly out-performs Oracle$_{\text{static}}$, reducing its energy consumption by 3–48% while satisfying accuracy constraints (36% in harmonic mean) and reducing its error rate by 9-66% while satisfying energy constraints (54% in harmonic mean).

Figure 8 shows a detailed comparison for the energy minimization task. The figure shows the range of performance under all requirement settings (i.e., the whiskers). ALERT not only achieves similar mean energy reduction, its whole range of optimization behavior is also similar to Oracle. In comparison, Oracle$_{\text{Static}}$ not only has the worst mean but also the worst tail performance. Due to space constraints, we omit the figures for other settings, where similar trends hold.

ALERT has more advantage over Oracle$_{\text{static}}$ on CPUs than on GPUs. The CPUs have more empirical variance than the GPU, so they benefit more from dynamic adaptation. The GPU experiences significantly lower dynamic fluctuation so the static oracle makes good predictions.

ALERT satisfies the constraint in 99.9% of tests for image classification and 98.5% of those for sentence prediction. For the latter, due to the large input variability (NLP1 in Figure 4), some input sentences simply cannot complete by the deadline even with the fastest DNN. There the Oracle fails, too.

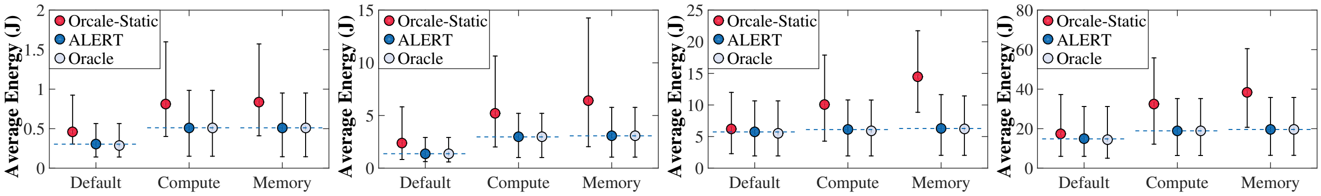
Note that, these Oracle schemes not only have perfect—and hence, impractical—prediction capability, but they also have no overhead. In contrast, ALERT is running on the same machines as the DNN workloads. *All results include ALERT's run-time latency and power overhead.*

Comparing with State-of-the-Art. For a fair comparison, we focus on ALERT$_{\text{Any}}$, as it uses exactly the same DNN candidate set as "Sys-only", "App-only", and "No-coord". Across all settings, ALERT$_{\text{Any}}$ outperforms the others.

The System-only solution suffers from not being able to choose different DNNs under different runtime scenarios. As a result, it performs much worse than ALERT$_{\text{Any}}$ in satisfying accuracy requirements or optimizing accuracy. For the former (left side of Table 4 and Figure 7), it creates accuracy violations in 68% of the settings as shown in Figure 7; for the latter (right side of Table 4 and Figure 7), although capable of satisfying energy constraints, it introduces 34% more error than ALERT$_{\text{Any}}$.

The Application-only solution that uses an Anytime DNN suffers from not being able to adjust to the energy requirements: it consumes 73% more energy in energy-minimizing tasks (left side of Table 4 and Figure 7) and introduces many energy-budget violations particularly under resource contention settings (right side of Table 4 and Fig. 7).

The no-coordination scheme is worse than both System- and Application-only. It violates constraints in both tasks with 69% more energy and 34% more error than ALERT$_{\text{Any}}$. Without coordination, the two levels can work at cross purposes; e.g., the application switches to a faster DNN to save energy while the system makes more power available.



(a) CPU1, Image Classification (b) CPU1, Sentence Prediction (c) CPU2, Image Classification (d) CPU2, Sentence Prediction

Figure 8: ALERT versus Oracle and Oracle$_{\text{Static}}$ on minimize energy task (Lower is better). (whisker: whole range; circle: mean)

Plat.	Work.	ALERT	Any	Trad	ALERT	Any	Trad
		Minimize Energy Task			Minimize Error Task		
CPU1	Idle	0.64	0.68	0.65^1	0.91	0.92	0.93
	Comp.	0.57	0.58	0.65^6	0.38	0.39	0.41
	Mem.	0.53	0.55	0.53^3	0.34	0.34	0.35
CPU2	Idle	0.93	0.88	0.95^1	0.68	0.68	0.69
	Comp.	0.59	0.57	0.60^4	0.58	0.57	0.59
	Mem.	0.38	0.37	0.40^8	0.23	0.24	0.32
GPU	Idle	0.97	0.99	0.95	0.90	0.92	0.89
	Comp.	0.97	1.01	0.96	0.89	0.92	0.89
	Mem.	0.96	0.97	0.95	0.32	0.34	0.32
Harmonic mean		0.66	0.66	0.67^3	0.47	0.48	0.50

Table 5: ALERT normalized average energy consumption and error rate to *Oracle*$_{\text{Static}}$ @ Sparse ResNet (Smaller is better)

5.3 Detailed Results and Sensitivity

Different DNN candidate sets. Table 5 compares the performance of ALERT working with an Anytime DNN (Any), a set of traditional DNN models (Trad), and both. At a high level, ALERT works well with all three DNN sets. Under close comparison, ALERT$_{\text{Trad}}$ violates more accuracy constraints than the others, particularly under resource contention on CPUs, because a traditional DNN has a much larger accuracy drop than an anytime DNN when missing a latency deadline. Consequently, when the system variation is large, ALERT$_{\text{Trad}}$ selects a faster DNN to meet latency and thus may not meet accuracy goals. Of course, ALERT$_{\text{Any}}$ is not always the best. As discussed in Section 3.5, Anytime DNNs sometimes have lower accuracy then a traditional DNN with similar execution time. This difference leads to the slightly better results for ALERT over ALERT$_{\text{Any}}$.

Figure 9 visualizes the different dynamic behavior of ALERT (blue curve) and ALERT$_{\text{Trad}}$ (orange curve) when the environment changes from Default to Memory-intensive and back. At the beginning, due to a loose latency constraint, ALERT and ALERT$_{\text{Trad}}$ both select the biggest traditional DNN, which provides the highest accuracy within the energy budget. When the memory contention suddenly starts, this DNN choice leads to a deadline miss and an energy-budget violation (as the idle period disappeared), which causes an accuracy dip. Fortunately, both quickly detect this problem and sense the high variability in the expected latency. ALERT switches to use an anytime DNN and a lower power cap. This switch is effective: although the environment is still unstable, the inference accuracy remains high, with slight ups and downs depending on which anytime output finished before the deadline. Only able to choose from traditional DNNs, ALERT$_{\text{Trad}}$ conservatively switches to much simpler and hence lower-accuracy DNNs to avoid deadline misses. This switch does eliminate deadline misses under the highly dynamic environment, but many of the conservatively chosen DNNs finish before the deadline (see the Latency panel), wasting the opportunity to produce more accurate results and causing ALERT$_{\text{Trad}}$ to have a lower accuracy than ALERT. When the system quiesces, both schemes quickly shift back to the highest-accuracy, traditional DNN.

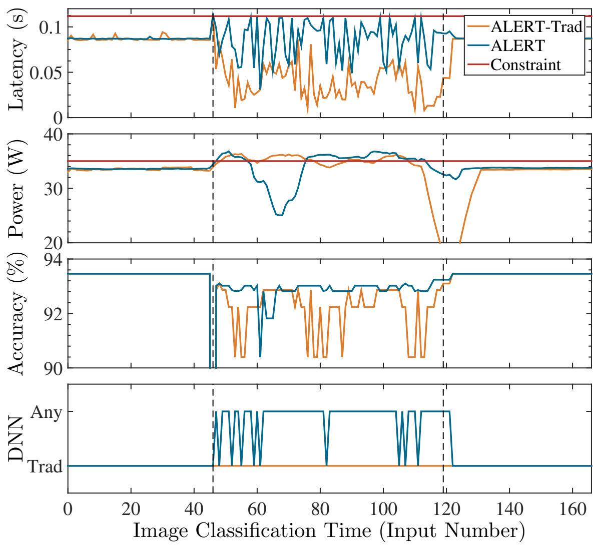
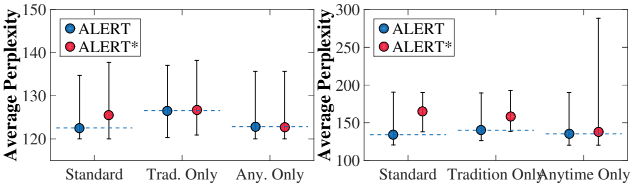


Figure 9: Minimize error rates w/ latency, energy constraints on CPU1. (Memory contention occurs from about input 46 to 119; Deadline: 1.25× mean latency of largest Anytime DNN in Default; power limit: 35W.)

Overall, these results demonstrate how ALERT always makes use of the full potential of the DNN candidate set to optimize performance and satisfy constraints.

ALERT probabilistic design. A key feature of ALERT is its use of not just mean estimations, but also their variance. To evaluate the impact of this design, we compare ALERT to an alternative design ALERT*, which only uses the estimated mean to select configurations.

Figure 10 shows the performance of ALERT and ALERT* in the minimize error task for sentence prediction. Here, ALERT (blue circles) always performs better than ALERT*. Its advantage is the biggest when the DNN candidates include both traditional and Anytime DNNs (i.e., the "Standard"



(a) Default Contention (b) Memory Contention

Figure 10: Minimize error for sentence prediction@ CPU1 (Lower is better). (whisker: whole range; circle: mean)

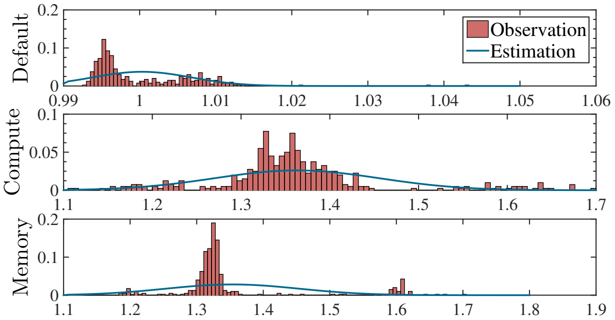


Figure 11: Distribution of ξ for image class. on CPU1.

in Figure 10). The reason is that traditional DNNs and Anytime DNN have different accuracy/latency curves, Eq. 3 for the former and Eq. 10 for the latter. ALERT* is much worse in distinguishing these two by simply using the mean of estimated latency to predict accuracy. ALERT also clearly outperforms ALERT* under memory contention with traditional DNN candidates, as ALERT's estimation better captures dynamic system variation. Overall, these results show ALERT's probabilistic design is effective.

Sensitivity to latency distribution. ALERT assumes a Gaussian distribution, but is designed to work for other distributions (see Section 3.6). As shown in Figure 11, the observed ξs (red bars) are indeed not a perfect fit for Gaussian distribution (blue lines), which confirms ALERT's robustness.

6 Related work

Past resource management systems have used machine learning [4, 51, 68, 69, 79] or control theory [32, 37, 44, 45, 62, 74, 93] to make dynamic decisions and adapt to changing environments or application needs. Some also use Kalman filter because it has optimal error properties [37, 44, 45, 62]. There are two major differences between them and ALERT: 1) prior approaches use the Kalman filter to estimate physical quantities such as CPU utilization [45] or job latency [37], while ALERT estimates a *virtual* quantity that is then used to update a large number of latency estimates. 2) while variance is naturally computed as part of the filter, ALERT actually uses it, in addition to the mean, to help produce estimates that better account for environment variability.

Past work designed resource managers explicitly to coordinate approximate applications with system resource usage [21, 31, 32, 46]. Although related, they manage applications *separately* from system resources, which is fundamentally different from ALERT's holistic design. When an environmental change occurs, prior approaches first adjust the application and then the system serially (or vice versa) so that the change's effects on each can be established independently [31, 32]. That is, coordination is established by forcing one level to lag behind the other. In practice this design forces each level to keep its own independent model and delays response to environmental changes. In contrast, ALERT's global slowdown factor allows it to easily model and update prediction about all application and system configurations simultaneously, leading to very fast response times, like the single input delay demonstrated in Figure 9.

Much work accelerates DNNs through hardware [3, 10–12, 19, 23, 24, 27, 30, 38, 43, 54, 58, 66, 73, 75, 83], compiler [9, 65], system [28, 53], or design support [25, 25, 26, 39, 42, 77, 81, 85]. They essentially shift and extend the tradeoff space, but do not provide policies for meeting user needs or for navigating tradeoffs dynamically, and hence are orthogonal to ALERT.

Some research supports hard real-time guarantees for DNNs [92], providing 100% timing guarantees while assuming that the DNN model gives the desired accuracy, the environment is completely predictable, and energy consumption is not a concern. ALERT provides slightly weaker timing guarantees, but manages accuracy and power goals. ALERT also provides more flexibility to adapt to unpredictable environments. Hard real-time systems would fail in the co-located scenario unless they explicitly account for all possible co-located applications at design time.

7 Conclusion

This paper demonstrates the challenges behind the important problem of ensuring timely, accurate, and energy efficient neural network inference with dynamic input, contention, and requirement variation. ALERT achieves these goals through dynamic and coordinated DNN model selection and power management based on feedback control. We evaluate ALERT with a variety of workloads and DNN models and achieve high performance and energy efficiency.

Acknowledgement

We thank the reviewers for their helpful feedback and Ken Birman for shepherding this paper. This research is supported by NSF (grants CNS-1956180, CNS-1764039, CNS-1764039, CNS-1514256, CNS-1823032, CCF-1439156), ARO (grant W911NF1920321), DOE (grant DESC0014195 0003), DARPA (grant FA8750-16-2-0004) and the CERES Center for Unstoppable Computing. Additional support comes from the DARPA BRASS program and a DOE Early Career award.

References

[1] Baidu AI. Apollo open vehicle certificate platform. Online document, http://apollo.auto, 2018.

[2] S. Akhlaghi, N. Zhou, and Z. Huang. Adaptive adjustment of noise covariance in kalman filter for dynamic state estimation. In *IEEE Power Energy Society General Meeting*, 2017.

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ISCA*, pages 1–13, 2016.

[4] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *CASES*, 2012.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.

[6] Giorgio C Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer, 2006.

[7] Aaron Carroll and Gernot Heiser. Mobile multicores: Use them or waste them. In *HotPower*, 2013.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594, 2018.

[10] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGPLAN Not.*, pages 269–284, 2014.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 2016.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO 47*, pages 609–622, 2014.

[13] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED*, 2010.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[16] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: An evaluation of the state of the art. *TPAMI*, 2011.

[19] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ISCA*, pages 92–104, 2015.

[20] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Mobicom*, 2018.

[21] Anne Farrell and Henry Hoffmann. MEANTIME: achieving both minimal energy and timeliness with approximate computing. In *USENIX ATC*, 2016.

[22] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry P Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *CVPR*, page 7, 2017.

[23] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Christos Kozyrakis. Draf: a low-power dram-based reconfigurable acceleration fabric. *ISCA*, pages 506–518, 2016.

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, pages 243–254, 2016.

[25] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[26] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *DATE*, pages 1474–1479, 2017.

[27] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *ISCA*, pages 27–40, 2015.

[28] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, pages 223–238, 2015.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[30] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Chang-Hong Hsu, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *MICRO*, pages 786–799, 2017.

[31] Henry Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *ECRTS*, pages 223–232, 2014.

[32] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *SOSP*, 2015.

[33] Henry Hoffmann and Martina Maggio. PCP: A generalized approach to optimizing performance under power constraints through resource management. In *ICAC*, pages 241–247, 2014.

[34] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*, 2019.

[35] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. In *CoRR*, 2017.

[36] C. Imes and H. Hoffmann. Bard: A unified framework for managing soft timing and power constraints. In *SAMOS*, pages 31–38, 2016.

[37] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *RTAS*, pages 75–86, April 2015.

[38] Animesh Jain, Michael A Laurenzano, Gilles A Pokam, Jason Mars, and Lingjia Tang. Architectural support for convolutional neural networks on modern cpus. In *PACT*, 2018.

[39] Shubham Jain, Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Pierce Chuang, and Leland Chang. Compensated-dnn: energy efficient low-precision deep neural networks by compensating quantization errors. In *DAC*, pages 1–6, 2018.

[40] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*, pages 253–266, 2018.

[41] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[42] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *ICS*, page 23, 2016.

[43] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, pages 1–12, 2016.

[44] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.

[45] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive resource provisioning for virtualized servers using kalman filters. *TAAS*, 2014.

[46] Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, 2013.

[47] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ICCPS*, pages 287–296, 2018.

[48] D. H. K. Kim, C. Imes, and H. Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *ICCPS*, 2015.

[49] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *arXiv preprint arXiv:1605.07648*, 2016.

[50] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX ATC*, June 2011.

[51] Benjamin C Lee and David Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. *ASPLOS*, 2008.

[52] Hankook Lee and Jinwoo Shin. Anytime neural prediction via slicing networks vertically. *arXiv preprint arXiv:1807.02609*, 2018.

[53] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *ASPLOS*, pages 751–766, 2018.

[54] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ISCA*, pages 369–381, 2015.

[55] Jun S Liu and Rong Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American statistical association*, 1998.

[56] ATLAS LS. What is simultaneous/conference interpretation? Online document, https://atlasls.com/what-is-simultaneousconference-interpretation/, 2010.

[57] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Automated control of multiple software goals using multiple actuators. In *FSE*, 2017.

[58] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, pages 14–26. IEEE, 2016.

[59] Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. Treebank-3 - linguistic data consortium. Online document, https://catalog.ldc.upenn.edu/LDC99T42, 1999.

[60] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. *TCCA*, 1995.

[61] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. *arXiv preprint arXiv:1703.06217*, 2017.

[62] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: learning control for predictable latency and low energy. In *ASPLOS*, 2018.

[63] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. *ASPLOS*, 2015.

[64] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[65] NVIDIA. Nvidia tensorrt: Programmable inference accelerator. Online document, https://developer.nvidia.com/tensorrt, 2018.

[66] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2015.

[67] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *SOSP*, 2017.

[68] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*, 2013.

[69] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO*, 2001.

[70] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil D. Dutt. SPECTR: formal supervisory control and coordination for many-core systems resource management. In *ASPLOS*, pages 169–183, 2018.

[71] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[72] S. Reda, R. Cochran, and A. K. Coskun. Adaptive power capping for servers with multithreaded workloads. *MICRO*, 2012.

[73] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*, page 18, 2016.

[74] Muhammad Husni Santriaji and Henry Hoffmann. Grape: Minimizing energy for gpu applications with performance requirements. In *MICRO*, 2016.

[75] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, page 17, 2016.

[76] N Silberman and Guadarrama. S. Tensorflow-slim image classification model library. Online document, https://github.com/tensorflow/models/tree/master/research/slim, 2016.

[77] Hyeonuk Sim, Saken Kenzhegulov, and Jongeun Lee. Dps: dynamic precision scaling for stochastic computing-based deep neural networks. In *DAC*, page 13, 2018.

[78] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[79] Srinath Sridharan, Gagan Gupta, and Gurindar S Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, 2013.

[80] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *ASE*, 2018.

[81] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *DAC*, 2017.

[82] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *CVPR*, 2016.

[83] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, page 4, 2011.

[84] Andreas Veit and Serge Belongie. Convolutional networks with adaptive inference graphs. In *ECCV*, 2018.

[85] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *ISLPED*, 2014.

[86] Chengcheng Wan, Henry Hoffmann, Shan Lu, and Michael Maire. Orthogonalized SGD and nested architectures for anytime neural networks. In *ICML 2020, to appear*.

[87] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. Alert: Accurate learning for energy and timeliness. *arXiv preprint arXiv:1911.00119*, 2020.

[88] Yan Wang, Zihang Lai, Gao Huang, Brian H Wang, Laurens van der Maaten, Mark Campbell, and Kilian Q Weinberger. Anytime stereo image depth estimation on mobile devices. *arXiv preprint arXiv:1810.11408*, 2018.

[89] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*, pages 8817–8826, 2018.

[90] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

[91] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *ATC*, pages 951–965, 2018.

[92] H. Zhou, S. Bateni, and C. Liu. S3dnn: Supervised streaming and scheduling for gpu-accelerated real-time DNN workloads. In *RTAS*, 2018.

[93] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. Cash: Supporting iaas customers with a sub-core configurable architecture. In *ISCA*, 2016.

NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems

Soroush Bateni and Cong Liu
The University of Texas at Dallas

Abstract

Deep neural networks (DNNs) used in computer vision have become widespread techniques commonly used in autonomous embedded systems for applications such as image/object recognition and tracking. The stringent space, weight, and power constraints seen in such systems impose a major impediment for practical and safe implementation of DNNs, because they have to be latency predictable while ensuring minimum energy consumption and maximum accuracy. Unfortunately, exploring this optimization space is very challenging because (1) smart coordination has to be performed among system- and application-level solutions, (2) layer characteristics should be taken into account, and more importantly, (3) when multiple DNNs exist, a consensus on system configurations should be calculated, which is a problem that is an order of magnitude harder than any previously considered scenario. In this paper, we present NeuOS, a comprehensive latency predictable system solution for running multi-DNN workloads in autonomous systems. NeuOS can guarantee latency predictability, while managing energy optimization and dynamic accuracy adjustment based on specific system constraints via smart coordinated system- and application-level decision-making among multiple DNN instances. We implement and extensively evaluate NeuOS on two state-of-the-art autonomous system platforms for a set of popular DNN models. Experiments show that NeuOS rarely misses deadlines, and can improve energy and accuracy considerably compared to state of the art.

1 Introduction

The recent explosion of computer vision research has led to interesting applications of learning-driven techniques in autonomous embedded systems (AES) domain such as object detection in self-driving vehicles and image recognition in robotics. In particular, deep neural networks (DNNs) with generally the same building blocks have been dominantly applied as effective and accurate implementation of image recognition, object detection, tracking, and localization towards enabling full autonomy in the future [60, 50]. For example, using such DNNs alone, Tesla has recently demonstrated that a great deal of autonomy in self-driving cars can be achieved [33]. Another catalyzer for the feasibility of DNN-driven autonomous systems in practice has been the advancement of fast, energy-efficient embedded platforms, particularly accelerator-enabled multicore systems such as the NVIDIA Drive AGX and the Tesla AI platforms [44, 22].

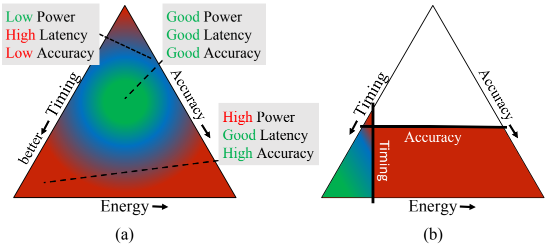


Figure 1: Ternary depiction of the 3D optimization space.

Autonomous systems based on embedded hardware platforms are bounded by stringent Space, Weight, and Power (SWaP) constraints. The SWaP constraints require system designers to carefully take into account energy efficiency. However, DNN-driven autonomous embedded systems are considered mission-critical real-time applications and thus, require predictable latency[1] and sufficient accuracy[2] (of the DNN output) in order to pass rigorous certifications and be safe for end users [46]. This causes a challenging conflict with energy efficiency since accurate DNNs require a tremendous amount of resources to be feasible and to be timing-predictable, and are by far the biggest source of resource consumption in such systems [5]. This usually results in less complicated (and less resource-demanding) DNN models to be designed and used in these systems, reducing accuracy considerably.

Fig. 1(a) shows a hypothetical three-dimensional space between latency, power, and accuracy mapped to a ternary plot [55] (where (Energy + Timing + Accuracy) has been normalized to 3). Each dot in Fig. 1(a) represents a configuration with a unique set of latency, power, and accuracy characteristics. The power consumption is usually

[1] Latency from each system component (including the DNNs) in AES will add up to the reaction latency between when a sensor observes an event and when the system externally reacts to that event, such as by applying the breaks in a self-driving vehicle. The faster a system reacts, the more likely it is for the system to avoid a disaster, such as an accident. However, policymakers might adopt a reasonable reaction time, such as 33ms or even 300ms [48, 27, 12, 14, 9, 4] as ”safe enough".

[2] We should mention here that there is currently no established standard to connect DNN accuracy to the safety of a particular system, such as DNNs in self-driving vehicles. In this paper, we assume the more accurate the DNN, the safer the system is.

adjusted at system-level via dynamic voltage/frequency scaling (DVFS) [5, 21]. The accuracy adjustment is done at application-level via DNN approximation configuration switching (see Sec. 4 for details). Note that both DVFS and DNN configuration adjustments would impact runtime latency. This figure highlights three configurations with various levels of latency, power consumption, and accuracy tradeoff that might or might not be acceptable given the current performance constraints. Choosing the best three-dimensional trade-off optimization point is a significant challenge given the vast and complex DVFS and accuracy configuration space.

Although all autonomous systems are required to be latency predictable in nature, the constraints on power and accuracy may vary based on the type of autonomous system (e.g., highly constrained power for drones and maximum accuracy requirement for autonomous driving). To illustrate one such variation, note Fig. 1(b), which shows a constraint on latency, and a constraint on accuracy imposed in the configuration space limiting the possible configurations considerably.

Challenges specific to DNN-driven AES. In addition to the aforementioned optimization problem, DNNs are constructed from layers, where each layer responds differently to DVFS changes and has unique approximation characteristics (as we shall showcase in Sec. 3.1). In order to meet a latency target with optimized energy consumption and accuracy, each layer requires a unique DVFS and approximation configuration, whereas existing approaches such as Poet [23] and JouleGuard [21] deal with DNNs as a black-box. Moreover, system-level DVFS adjustments and application-level accuracy adjustments happen at two separate stages. Without smart coordination, the system might fall in a negative feedback loop, as we shall demonstrate in Sec. 3.2. This coordination needs to happen at layer boundaries, making the problem at least an order of magnitude harder than previous work.

Furthermore, existing techniques mostly focus on single-tasking scenarios [5, 3, 20] whereas AES generally require multiple instances of different DNNs. As we shall motivate in Sec. 3.3 using a real-world example, these DNNs need to communicate and build a cohort on a layer-by-layer basis to avoid greedy and inefficient decision-making. Moreover, system-level and application-level coordination in this multi-DNN scenario is much harder than isolated processes considered in previous work.

Finally, existing approaches [13, 5] optimize latency performance on a best-effort basis (e.g., by using control theory) that can overshoot a latency target (as demonstrated in Sec. 3.2). A better solution should include proven real-time runtime strategies such as LAG analysis [51].

Contribution. In this paper, we present NeuOS[3], a comprehensive timing-predictable system solution for multi-DNN workloads in autonomous embedded systems. NeuOS can manage energy optimization and dynamic accuracy adjustment for DNNs based on specific system constraints via smart coordinated system- and application-level decision-making.

NeuOS is designed fundamentally based on the idea of multi-DNN execution by introducing the concept of cohort, a collective set of DNN instances that can communicate through a shared channel. To track this cohort, we address how latency, energy, and accuracy can be measured and propagated efficiently in the multi-DNN cohort.

Besides the fundamental goal of providing latency predictability (i.e., meeting deadlines for processing each DNN instance), NeuOS addresses the challenge of balancing energy at system level and accuracy at application level for DNNs, which has never been addressed in literature to the best of our knowledge. Balancing three constraints at various execution levels in the multi-DNN scenario requires smart coordination 1) between system level and application level decision making, and 2) among multiple DNN instances.

Towards these coordination goals, we introduce two algorithms in Sec. 4.2 that are executed at the layer completion boundary of each DNN instance: one algorithm that can predict the best system-level DVFS configuration for each DNN member of the cohort to meet deadline and minimize power for that specific member in the upcoming layer, and one algorithm that decides what application level approximation configuration is required for others if any one of these system-level DVFS decisions were chosen. These two algorithms effectively propagate all courses of action for the next layer in order to meet the deadline. Based on these two algorithms, we propose an optimization problem in Sec. 4.3 that can decide the best course of action depending on the system constraint, and minimize system overhead. This method is effective because 1) it introduces an identical decision-making among all DNN instances in the cohort and solves the coordination problem between system-level and application-level decision making, and 2) provides adaptability to three typical scenarios imposing different constraints on energy and accuracy.

Implementation and Evaluation. We implement a system prototype of NeuOS and extensively evaluate NeuOS using popular image detection DNNs as a representative of convolutional deep neural networks used in AES. The evaluation is done under the following conditions:

- **Extensible in terms of architecture.** We fully implement NeuOS using a set of popular DNN models on two different platforms: an NVIDIA Jetson TX2 SoC (with architecture designed for low overhead embedded systems), and an NVIDIA AGX Xavier SoC (with architecture designed for complex autonomous systems such as self-driving cars).
- **Multi-DNN scenarios.** We ensure that our system can trade-off and balance multiple DNNs in all conditions by testing NeuOS under three cohort sizes: a small 1-process, a medium 2-4 process, and a large 6-8 process.

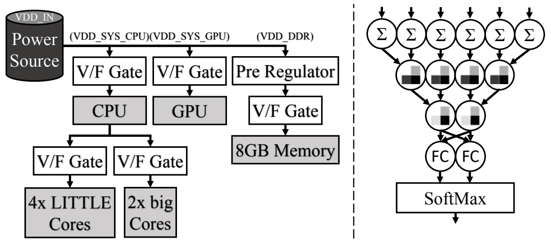
[3] The latest version of NeuOS can be found at https://github.com/Soroosh129/NeuOS.

- **Latency predictability.** We extensively compare NeuOS to six state-of-the-art solutions in literature, and find that NeuOS rarely misses deadlines under all evaluated scenarios, and can improve runtime latency on average by 68% (between 8% and 96% depending on DNN complexity) on TX2, by 40% on average (between 12% and 89%) on AGX, and by 54% overall.
- **Versatility.** NeuOS can be easily adapted to the following three constraint scenarios:
  - **Balanced energy and accuracy.** Without any system constraints given, NeuOS is proved to be energy efficient while sacrificing an affordable degree of accuracy, improving energy consumption on average by 68% on TX2, by 40% on average on AGX, while incurring an accuracy loss of 21% on average (between 19% and 42%).
  - ***Min energy.*** When energy is constrained to be minimal, NeuOS is able to sacrifice accuracy a small amount (at most 23%) but further improve energy consumption by 11% over the general unrestricted case, while meeting the latency requirement.
  - ***Max accuracy.*** When accuracy is given as a constraint, NeuOS is able to improve accuracy by 10% on average compared to balanced case, but also sacrifices energy by only a small amount, increasing by 23% on average.

2 Background

DVFS space in autonomous systems. The trade-off between latency and power consumption is usually achieved via adjustments to frequency and/or voltages of hardware components. A software and hardware technique typical of modern systems is DVFS. Through DVFS, system software such as the operating system or hardware solutions can dynamically adjust voltage and frequency. To understand this technique better, consider Fig. 2(a), showing the components of a Jetson TX2, which contains a Parker SoC with a big.LITTLE architecture with 2 NVIDIA Denver big cores and 4 ARM Cortex A53 LITTLE cores. The Parker SoC also contains a 256-core Pascal-architecture GPU. The TX2 module also contains 8 GB of shared memory (the Jetson AGX Xavier also used in Sec. 5 has a more advanced Xavier SoC with 8 NVIDIA "Carmel" cores, a 512 Volta-architecture GPU, and 16GB of shared memory). Each component includes a voltage/frequency (V/F) gate that can be adjusted via software. The value for frequency and voltage for each component forms a unique topple, called a DVFS configuration throughout this paper.

DNN and its approximation techniques. Fig. 2(b) depicts a simplified version of a Deep Neural Network (DNN). Neurons are the basic building blocks of DNNs. Depending on the layer neurons belong to, they perform various different operations. A DNN may contain multiple layers of different types, such as the convolutional and the normalization layers, which are connected via their inputs and outputs.



(a) Jetson TX2 V/F Structure (b) An example DNN

Figure 2: DVFS configuration space and DNN structure.

DNNs by nature are approximation functions [36]. DNNs are trained on a specific training set. After training, accuracy is measured by using a test data set, set aside from the training set and measuring the accuracy (e.g., top-5 error rate–comparing the top 5 guesses against the ground truth). The accuracy of the overall DNN can be adjusted by manipulating the layer parameters.

A rich set of DNN approximation techniques have been proposed in the literature and adopted in the industry [17, 58, 24, 29, 43, 56, 7, 18]. Such techniques aim at reducing the computation and storage overhead for executing DNN workloads. An example technique to provide approximation for convolutional layers is Lowrank [45], which performs a lowrank decomposition of the convolution filters. In our implementation, dynamic accuracy adjustment or "hot swapping" layers will refer to applying the lowrank decomposition to the upcoming layers before their execution. Note that applying such approximation adjustments on the fly is possible because the generated pair of layers have the exact combined input and output dimensions. Moreover, this adjustment is only possible for future layers at each layer boundary.

Measuring Accuracy. The approximation on the fly will affect the final accuracy. Due to the dynamic nature of this adjustment, the exact value of accuracy measurement using traditional methodology is impractical. Most related work thus incorporate an alternative scoring method [3], where the system will deduce the accuracy score accordingly if certain approximation techniques are to be applied to the next layer. In our method, we assume a perfect score for the original DNN, and switching to the lowrank approximation of any layer will reduce the score by a set amount. For example, running AlexNet in its entirety will result in a score of 100. If we swap a convolutional layer with a lowrank version of that layer, the overall accuracy will be affected by some amount (e.g., 1 in our method), thus yielding a lower score (e.g., 99 under the scoring method). Therefore, the score is always relative to the original DNN configuration and not related to the absolute value of accuracy on a particular dataset. This method of keeping relative accuracy is still invaluable to maximizing accuracy in a dynamic runtime environment but cannot be used to calculate the exact accuracy loss.

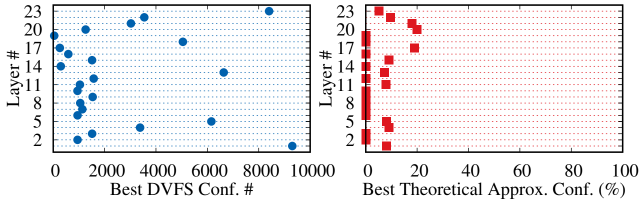


Figure 3: Calculated best system level DVFS configuration and best application level theoretical approximation configuration for AlexNet on Jetson TX2 in order to meet a 12ms deadline (0 means no approximation).

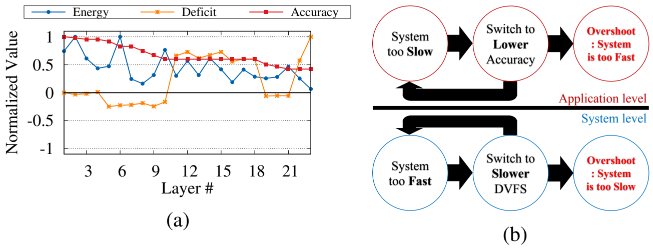
3 Motivation

In this section, we lay out several motivational case studies to understand the challenges that exist for DNNs, and gain insights on why existing approaches (or naively extended ones) may fail under our problem context.

3.1 Balancing in two-dimensional Space

The trade-off to meet a specified latency target while maximizing accuracy is done in a 2-dimensional space by choosing an approximation configuration for the application. Similarly, the 2-dimensional trade-off between energy and latency is done by changing an optimal DVFS configuration. Traditional control-theory based solutions treat the entire application as a black-box, and decide on what DVFS or approximation configuration should be chosen every few iterations of that specific application [21, 20]. However, treating DNNs as a blackbox does not yield the most efficient results. Fig. 3 left hand shows the best DVFS configuration for each layer of AlexNet among all possible DVFS configurations for a Jetson TX2 in terms of energy consumption. The y-axis is the layer number for AlexNet, and the x-axis is the DVFS configuration index, partially sorted based on frequency and activated core counts. The dots show the configuration that has the absolute minimum energy consumption. As is evident, each layer has a different optimal DVFS configuration. More interestingly, we observe a non-linearity where sometimes faster DVFS configurations have lower energy consumption. This is due to the massive parallelism of GPUs, where increasing the frequency by 2x for example can yield a 10-fold improvement in performance, which outweighs the momentary increase in energy consumption. Fig. 3 right hand shows the best theoretical approximation configurations required for each layer of AlexNet in order to meet a 12ms deadline[4]. As is evident in the figure, each layer requires a different approximation configuration for optimal results.

Thus, the DNN must somehow become transparent to the system, conveying layer-by-layer information in order to make the correct decisions. This can make the decision space in the 2-dimensional space at least an order of magnitude harder (e.g., AlexNet has 23 layers) since every layer must be considered for each execution of the DNN application.



(a)

(b)

Figure 4: Negative feedback loop between an application-level solution and a system-level solution.

Observation 1: Layer-level trade-off makes the problem an order of magnitude harder than ordinary blackbox techniques.

3.2 Balancing in three-dimensional Space

Balancing energy/latency and accuracy/latency in isolation can be naive, and lead to unnecessary consumption of energy or reduced accuracy. Fig. 4a shows a similar experiment to Sec. 3.1, but both the system and application (Alexnet) are employed at the same time without any coordination. The goal of both solutions is to reach a 20ms deadline (by using latency deficit, LAG, as a guide (Sec. 4.2)). In the case of AlexNet, the system-level DVFS adjustment can be enough to meet the desired deadline. In an ideal scenario, only energy is adjusted slightly until AlexNet is not behind schedule. However, as is evident in the figure, normalized energy consumption and accuracy for each layer are both decreased continuously and dramatically. This is due to an unwanted negative loop, where a negative deficit (indicating that the system is behind schedule) has resulted in the application-level solution switching to a lower approximation configuration. Because these configurations are discrete, as we shall discuss in Sec. 4.2, the deficit will overshoot (at around layer 10) and becomes positive (meaning the system is ahead of schedule). The system-level solution would see this deficit as a headroom to reduce energy consumption, and in the case of Fig. 4a, has turned the positive deficit into a small negative at around layer 18. This cycle (as depicted in Fig. 4b) is repeated until the minimum approximation configuration is reached. This result is extremely undesirable in accuracy-sensitive applications such as autonomous driving (but can be okay for energy sensitive applications such as remote sensing). Thus, a feasible solution would be for the system and application to communicate, and make decisions based on given constraints for an application based on given constraints. This communication should be done at the granularity of layers, which makes the problem extra hard.

Observation 2: Trade-off in a 3-dimensional latency, energy, and accuracy optimization space is a significant challenge due to both system constraints as well as lacking harmony between application-level and system-level solutions.

3.3 Balancing for Multi-DNN Scenarios

To the best of our knowledge, no existing approach deals with multiple DNN instances in a coordinated manner.

[4] Please see Sec. 4.2 for more details on how this is calculated.

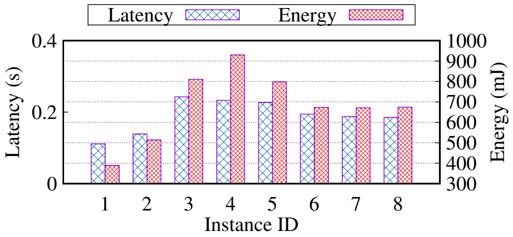


Figure 5: Energy consumption and execution time of running 8 instances of Resnet-50 on a Jetson TX2 under PredJoule.

Straightforwardly extending single-tasking latency/energy trade-off approaches, such as PredJoule [5], to multi-tasking scenarios would only result in decision-making that is local and greedy, based on locally measured variables. To showcase why coordination in this additional dimension is a key issue, examine Fig. 5, which shows the latency and energy consumption for running 8 DNN instances together averaged over 20 iterations under PredJoule on a Jetson TX2. We chose PredJoule because in our experiments, it outperformed all other existing solutions on exploring the 2D tradeoff between latency and energy for DNNs. The left (right) y-axis in Fig. 5 depicts the latency (energy consumption) in seconds (miliJoules) for each instance. As is evident in the figure, the DVFS management is greedy, resulting in instances 1 and 2 having relatively good latency and energy consumption. This greediness has pushed the rest of the DNN instances into unacceptable latency range (which is above 150ms for ResNet-50) because the chosen DVFS configuration at each layer boundary has been mostly beneficial only to the current layers of DNN instance 1 and 2. Moreover, the distribution of timing and energy consumption is not even across all instances because of the same reason. This disparity is the result of an uncoordinated system solution that chooses DVFS configurations greedily based on local variables.

Observation 3: In addition to the 2D and 3D complexities of solving the latency/accuracy/energy trade-off, a complete system solution must also accommodate for Multi-DNN scenarios, which are inherently more complicated to model and predict than single-DNN scenarios. The case studies also imply that naive extensions on existing single-DNN 2D solutions may fail in multi-DNN cases because they make greedy decisions based on local variables without coordination towards being globally optimal.

4 System Design

4.1 NeuOS Overview

To optimize the three-dimensional tradeoff space at the layer granularity, two basic research questions need to be answered first: 1) *how* to define and track the values of the three performance constraints in the system, and 2) *what* target should be imposed for optimizing each constraint.

For the first research question, we define a value of LAG (defined in Sec. 4.2, as a measurement of how far behind the DNN is compared to an ideal schedule that meets the relative deadline D), which tracks the progress of DNN execution at layer boundaries, P for energy consumption (in mJ) for each layer, and a variable X to reflect accuracy. We choose to track LAG at runtime instead of using an end-to-end optimization because it is more practical due to two reasons: 1) in a multi-DNN scenario, predicting the overlap between different DNN instances (and thus coordinating an optimal solution) cannot be done offline without making unrealistic assumptions, such as synchronized release times, and, 2) LAG is especially useful in a real system since it can account for outside interference, such as interference by other processes in the system, whereas an end-to-end optimization framework could miss the latency target. Moreover, as we shall discuss in Sec. 4.2, the value of P can be inferred by LAG in our design as these two variables fundamentally depend on the runtime DVFS configuration. Thus, the essential variables to track the status of a DNN execution can be simplified to $\{LAG, X\}$. Since we are dealing with a multi-DNN scenario, each DNN instance will have its own set of these variables. To know the collective status of the system, each DNN instance will put its variables in a shared queue.

In order to answer the second question regarding what optimization targets should be imposed on the system, we focus on the following three typical scenarios (expanded on in Sec. 4.3) that entail different performance constraints:

- **Min Energy** (M_P) is when NeuOS is deployed on an embedded system with a critically small energy envelope. Thus, the system should minimize energy without sacrificing too much accuracy. This scenario is motivated by applications seen in extremely power-limited systems such as drones, robotics, and a massive set of internet-of-thing devices.
- **Max Accuracy** (M_A) is when NeuOS is deployed on a system that has limited energy but accuracy is of utmost importance. Thus, the system should try to maximize accuracy without losing too much energy. This scenario is motivated by CPS-related applications such as autonomous driving.
- **Balanced Energy and Accuracy** (S) describes a more general, flexible scenario when the system is limited by both energy consumption and accuracy requirements, but no priority is given to either. Thus, the system should try to balance energy consumption and accuracy.

With the given scenarios and the values of $\{LAG, X\}$ at hand, we can answer the two key research questions presented in our motivation: 1) how to coordinate in a multi-DNN scenario such that the overall system is balanced and can meet the performance constraints, and, 2) how to efficiently tradeoff between latency, energy, and accuracy given the complexity of the problem space and how to prevent the negative feedback loop discussed in Sec. 3.2?

Design overview. Fig. 6 shows the overall design of NeuOS

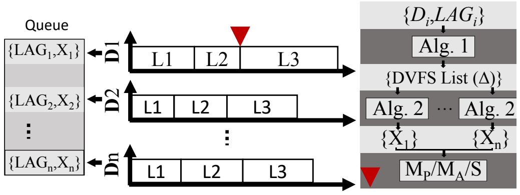


Figure 6: Design Overview

around $\{LAG, X\}$. The left side depicts the shared queue among multiple DNN instances. In the middle, a simple example of n concurrently running DNN instances each with three layers is shown. NeuOS makes runtime decisions on DVFS and DNN approximation configuration adjustments at layer boundaries, i.e., whenever a layer of a DNN instance completes. This is beneficial not only because applying approximation on-the-fly is possible only at layer boundaries, but in terms of overhead as well (as proved by our evaluation).

As illustrated in the figure, at the boundary between layers L2 and L3 of the first DNN instance, NeuOS is going through the process of decision-making which contains several steps. The first step is Alg.1, which senses the last known value of LAG for each DNN instance. Alg. 1 decides what DVFS configuration (at system level) is best for each instance in order to meet their deadlines D, outputting a list of potential DVFS configurations (Δ), where each member of the list corresponds to a DNN instance. In the next step, the list of potential DVFS configurations are fed into Alg.2, which predicts what approximation $\{X_i\}$ (at application level) would be required for other DNN instances to meet the deadline if the DVFS configuration for any one of the DNN instances is applied. Thus, Alg. 1 and Alg. 2 in tandem discover all possible courses of action the system can take to meet the deadline. However, at this point, no decision has been made on what DVFS configuration or accuracy configuration should be chosen for the system, because that depends on the given system constraint. This problem is inherently an optimization problem of finding the best possible choice in the propagated configuration space. We present this optimization problem formally in Sec. 4.3, where depending on broad scenarios, a particular setting is chosen for the next period of execution. In the last step of NeuOS, the system chooses one of these possibilities based on the scenario involved.

4.2 Coordinated System- and Application-level Adjustments

In this section, we expand on how runtime LAG is measured, how it relates to energy consumption, how accuracy X is calculated, and how the two developed algorithms take advantage of these two measurements to discover all possible choices the system can make efficiently in order to reduce the LAG to zero and meet the deadline.

LAG. We quantify the relationship between the partial execution time at time t of DNN instance i (e_i) and its relative deadline D_i as a form of LAG [51], denoted by LAG_i. LAG_i is a local variable (that can be updated at layer boundaries) for each DNN instance that keeps track of how far ahead or how far behind the DNN instance is compared to the deadline at time t. LAG_i is calculated as:

$$LAG_i(t, L_i(t)) = \sum_{l \in L_i(t)} (d_l - e_l), \tag{1}$$

in which $L_i(t)$ is the list of the layers of instance i that have completed by time t. For layer $l \in L_i(t)$, d_l and e_l depict the sub-deadline for layer l and the recorded execution time for layer l, respectively. NeuOS keeps track of e_l by measuring the elapsed time between each layer. Moreover, we use the proportional deadline method [38] to devise sub-deadlines for each layer based on D_i, the relative (end-to-end) deadline of DNN instance i, in which the subdeadline d_l for layer l is calculated as:

$$d_l = (e_l / \sum_{x \in L_i} (e_x)) \cdot D_i, \tag{2}$$

where $\sum_{x \in L_i} (e_x)$ denotes the execution time of DNN i. The proportional nature of sub-deadlines means that they only need to be calculated once for the lifetime of a given DNN instance on a platform.

Each DNN instance i would broadcast LAG_i among all instances via the shared queue. Thus, LAG_i would reflect the last known status of DNN instance i up to the last executed layer. We call the collection of LAG from all instances the LAG cohort, and we denote it by Φ. At completion of a DNN instance, a special message is sent to the cohort so that every DNN instance in the system is aware of their exit.

Based on the LAG cohort, the DNN instances can make decisions on accuracy and DVFS. A cohort will be perfect if every LAG within it is 0, or $\forall LAG_i \in \Phi, LAG_i = 0$. This means that all layers have exactly finished by their sub-deadline so far. Thus, the system has reasons to believe that the DNN instances will exactly finish by the deadline and do not require a faster DVFS or an approximation configuration, saving energy and accuracy in the process.

Since LAG indicates how far behind ($LAG < 0$) or ahead ($LAG > 0$) each DNN is, the DVFS and the approximation configuration need to be adjusted to run faster or slower accordingly. However, energy consumption and accuracy constraints must also be considered. We discuss each next.

System-level DVFS adjustment. At system-level, the question is which DVFS configuration is the best given the state of Φ to minimize energy consumption while reducing LAG to zero? The answer would vary between different DNN instances in the cohort, as they exhibit different LAGs. Moreover, different layers react differently to DVFS adjustments.

Alg. 1 is responsible for finding the best DVFS configuration for each DNN instance in the cohort. Alg. 1 takes as input the LAG cohort Φ and a SpeedUp/PowerUp table for the current layer of each DNN instance i. The structure of

Algorithm 1 Δ Calculator.

```
Input: $\Phi$        ▷ Progress Cohort
Input: SpeedUp/PowerUp[]        ▷ The SpeedUp/PowerUp table of DNNs.
Output: $\Delta$

1: function RETURN$\Delta(\Phi)$
2:     for $LAG_i$ in $\Phi$ do
3:         $S_{P_i} \leftarrow \frac{D_i + LAG_i}{D_i}$.
4:         $\delta_i \leftarrow$ LookUp(SpeedUp/PowerUp[$S_{P_i}$])
```

Table 1: SpeedUp/PowerUp and SpeedUp/Accuracy tables.

(a) SpeedUp/PowerUp for a layer of DNN instance i.

DVFS Configuration(δ)	SpeedUp	PowerUp
1	1x	1x
2	2.1x	2x
3	2.8x	1.5x

(b) SpeedUp/Accuracy.

X	SpeedUp
81%	1x
71%	1.8x
59%	2.5x

the SpeedUp/PowerUp table is depicted in Table 1a. The first column of Table 1a is the index for all the possible DVFS configurations in the system. The second column indicates how fast each DVFS configuration is in the worst case scenario compared to the baseline DVFS configuration (baseline is usually chosen to be the slowest configuration). The third column indicates how much power that DVFS configuration will consume relative to baseline.

Storing relative speedup and powerup values (instead of absolute measurements) is useful for looking up the table. In Alg. 1, given a LAG_i (line 2) and a relative deadline D_i for DNN instance i, the required speedup (denoted as S_i) could be directly calculated as (line 3):

$$S_P = \frac{D_i + LAG_i}{D_i}, \tag{3}$$

in which S_P is the speedup (or slowdown) value calculated as the relationship between the current projected execution time ($D_i + LAG_i$) and the ideal execution time (D_i). Since LAG can be negative or positive, the value of S_P can indicate a slowdown or speedup, where the slowdown is a way to conserve energy, which is the goal of NeuOS. The LookUp procedure (line 4) would then find the closest DVFS configuration that matches the speedup (or slowdown) in relation to the current configuration.

For our Alg. 1 to operate, we prepare a structure such as Table 1a for all DNN instances in a hashed format[5]. The LookUp procedure would then directly find a bucket by using the SpeedUp as an index. The output of Alg. 1 is a set $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$, in which δ_i is the ideal DVFS configuration for DNN instance i in order to meet the deadline. Imagine we ultimately decide that $\delta_c \in \Delta$ is the best DVFS configuration for the next scheduling period. A very interesting question would be that, what is the effect of applying δ_c on other DNN instances $i \neq c$? The speedup of δ_c for other DNN instances can be calculated by using δ_c as the lookup key in their corresponding SpeedUp/PowerUp table. But what if this speedup does not reduce LAG_i to zero? To solve this problem, we next present the algorithm that calculates the application-level approximation required to reduce LAG_i to zero given a DVFS configuration $\delta_c \in \Delta$.

Algorithm 2 X_i Calculator.

```
Input: $\Delta$        ▷ Potential DVFS list.
Input: SpeedUp/Accuracy[]        ▷ The SpeedUp/Accuracy table of DNNs.
Input: SpeedUp/PowerUp[]        ▷ The SpeedUp/PowerUp table of DNNs.
Output: $X[][]$        ▷ The accuracy list for each DNN instance for each $\delta$

1: function RETURN$X_i(\Delta)$
2:     for $\delta_c$ in $\Delta$ do
3:         for $i = 0$ to $i < |\Delta|$ do
4:             $S_{A_i} \leftarrow \frac{S_{P_i}(\delta_c) \cdot (D_i + LAG_i)}{D_i}$
5:             $X[c][i] \leftarrow$ LookUp(SpeedUp/Accuracy[$S_{A_i}$])
```

Application-level accuracy adjustment. Alg. 2 portrays the procedures to calculate the required approximation for the upcoming layers of all DNN instances based on a DVFS configuration. If the instance i is behind the ideal schedule by LAG_i, with a relative deadline of D_i, and if the chosen DVFS configuration is δ_c, the remaining required speedup can be calculated as follows (line 4):

$$S_{A_i}(\delta_c) = \frac{S_{P_i}(\delta_c) \cdot (D_i + LAG_i)}{D_i}, \tag{4}$$

in which $S_{A_i}(\delta_c)$ is the required speedup (or slowdown) via approximation for DNN instance i when DVFS configuration δ_c is chosen, and $S_{P_i}(\delta_c) \cdot (D_i + LAG_i)$ is the new projected execution time of DNN instance i. The value of S_{A_c}, the speedup from accuracy for the chosen DVFS configuration, should always be zero or less than zero since by definition, δ_c is the ideal DVFS configuration for c and requires no additional speedup from approximation.

The value of S_{A_i} is then used as a lookup key to a new table, called the SpeedUp/Accuracy table, depicted in Table 1b. Table 1b stores the relative worst case execution times for each layer's approximation configuration. We index each row by X, which is the value of the total accuracy of that configuration[6]. Note that the exact value of X has no effect in the algorithm and what matters is the relative order in Table 1b (i.e., the lower we go down the table, the lower the relative accuracy). The output of Alg.2 is the row index in the SpeedUp/Accuracy table sufficient to meet the deadline for all DNN instances except c. We denote this index for layer k of DVFS configuration i as X_i^k. This value is then broadcasted in the accuracy cohort and indicates the application-level

[5] Our hashing is custom, and hashes the relationship between SpeedUp and PowerUp. This method relies on partially sorting the DVFS configuration space. You can find the latest hashing code at https://git.io/Jfogq

[6] Each row could be indexed by any measure. However, indexing with X has benefits in overhead reduction for the LookUp procedure in Alg. 2 because it can be more easily hashed.

configuration chosen for the next immediate layer of the corresponding DNN instance.

The remaining question is that which δ_c should be chosen. We answer this question next.

4.3 Constraints and Coordination

The combination of Alg. 1 and Alg. 2 produces a list of potential DVFS configurations Δ, and for each DVFS configuration in Δ, a corresponding list of required approximations for all DNN instances in the cohort if that DVFS configuration were to be applied. Such a scenario can be visualized as a decision tree. The remaining question of our design would be which path to go down to in order to have a perfect *LAG* cohort. As discussed in Sec. 3.2, the requirements on energy and accuracy can vary depending on specific scenarios. We present the following three approaches based on the three scenarios defined in Sec. 4.1, i.e., minimum energy (M_P), maximum accuracy (M_A), and balanced energy and accuracy (S).

Min Energy. This approach aims at minimizing power usage at the cost of accuracy. To choose the best DVFS configuration in the DVFS candidate set Δ, we should look at the corresponding $S_{P_i}(\delta_c), \delta_c \in \Delta$ values in the SpeedUp/PowerUp table and choose the δ_c that has the smallest PowerUp value for that corresponding DNN instance, namely:

$$\delta_c = \{\delta_i \in \Delta \mid PowerUp_i(\delta_i) \leq PowerUp_i(\delta_x), \forall \delta_x \in \Delta\}, \quad (5)$$

in which $PowerUp_i(\delta_i)$ is extracted from the SpeedUp/PowerUp table of DNN instance i. Note that in our experience, the values of PowerUp can be non-linear in relation to SpeedUp, and hence, a comprehensive search as noted above is required. Then, using Alg. 2, the accuracy cohort can be calculated and broadcasted based on the projected new execution times. Even though this approach has the best power consumption, it will not have the best accuracy since many processes will most likely not meet the deadline without significant loss of accuracy, since the speedup from DVFS alone will likely not make up for the vast majority of the progress values in the cohort.

Max Accuracy. In this method, our system chooses the DVFS configuration δ_c in such a way that:

$$\delta_c = \{\delta_i \in \Delta \mid \sum(S_{Aj}(\delta_i)) \leq \forall \sum(S_{Aj}(\delta_x \in \Delta)), \; \forall \text{ DNN instance j in cohort}\}, \quad (6)$$

in which $\sum S_{Aj}(\delta_i)$ is the sum of all the required speedups from approximation (S_{Ai}) for configuration δ_i, and $\leq \forall \sum(S_{Aj}(\delta_x \in \Delta))$ is indicating that the sum of approximation-induced speedup for the chosen δ_c should be less than or equal any other sum of approximation values for other $\delta_x \in \Delta$ (this indirectly ensures minimized accuracy loss).

Statistical Approach for Balanced Energy and Accuracy. To achieve balanced energy and accuracy, we propose a statistical approach that checks the state of Δ and the projected accuracy cohort in statistical terms to make a decision. The calculation of S_{P_i} and S_{A_i} (which depends on S_{P_i}) resemble the form of Bivariate Regression Analysis (BRA) [57], in which:

$$S_{A_i} = S_{P_i} \cdot \frac{D_i + LAG_i}{D_i} + 0, \quad (7)$$

in which, $\frac{D_i + LAG_i}{D_i}$ is called the influence of S_{P_i} on the required approximation. To measure this influence, we first calculate

$$I = \sum_{i=0}^{i=n-1} \frac{D_i + LAG_i}{D_i}, \quad (8)$$

in which I is the collective influence of LAG on approximation. If the value of I is high, it means that the accuracy can be more adversely affected by a low value of DVFS-induced speedup(S_{P_i}). Similarly, a low value of I means that the accuracy can remain minimal even with a low value for DVFS-induced speedup. We simplify our decision making by dividing the *LAG* cohort Φ into three groups based on how big or small the value of *LAG* is. The boundary for the intervals is calculated using:

$$Boundary = \frac{\max\{\Phi\} - \min\{\Phi\}}{3}. \quad (9)$$

The three groups $G1[0...Boundary]$, $G2[Boundary...2 \cdot Boundary]$, and $G3[2 \cdot Boundary...3 \cdot Boundary]$ are then formed, and the ultimate DVFS configuration is chosen as:

$$\delta_c = \begin{cases} median(G1) & if(I < t) \\ median(G2) & if(t < I < 1+t) \\ median(G3) & if(I > 1+t) \end{cases}, \quad (10)$$

in which, t is a threshold for I, set to the standard deviation σ of the set I. However, t can be chosen by the system designer to indicate a requirement on power consumption and accuracy. A small value for t will push the system towards faster DVFS configurations and vice versa.

Discussion on choosing modes and safety. We would like to conclude our design by a discussion on which modes to choose and the safety concern it might entail. Our stand from a system perspective is to design a flexible system architecture that can adapt to various external needs. Where absolute mission-critical applications are concerned, we offer Max Accuracy. Nonetheless, depending on the safety requirement, our Balanced approach might be good enough with the proper threshold t even for applications such as self-driving vehicles. However, choosing a mode dynamically at runtime or statically for a particular system offline has more to do with the certification standards (which are in their preliminary stages for self-driving vehicles) as well as the requirement on maximum reaction time and accuracy. Thus, we believe the decision should be relegated to an external policy controller [54, 31, 6, 34, 52]

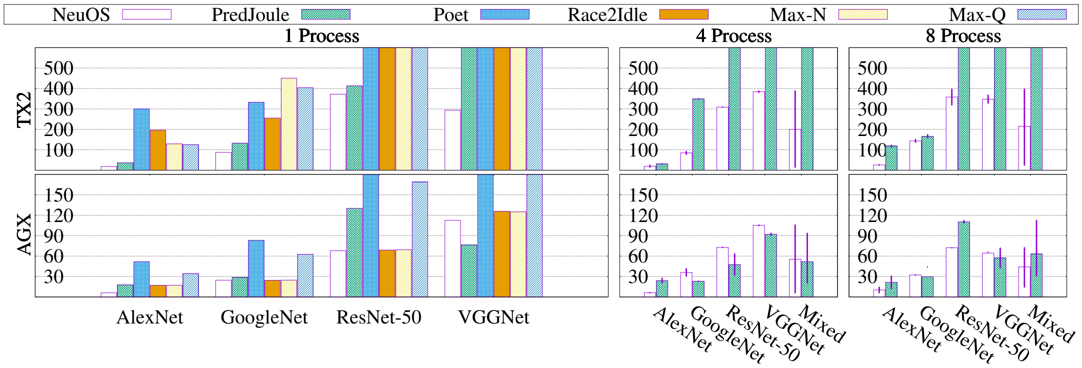


Figure 7: Energy under various methods (in mJ) for 1, 4, and 8 instances of 4 DNN models.

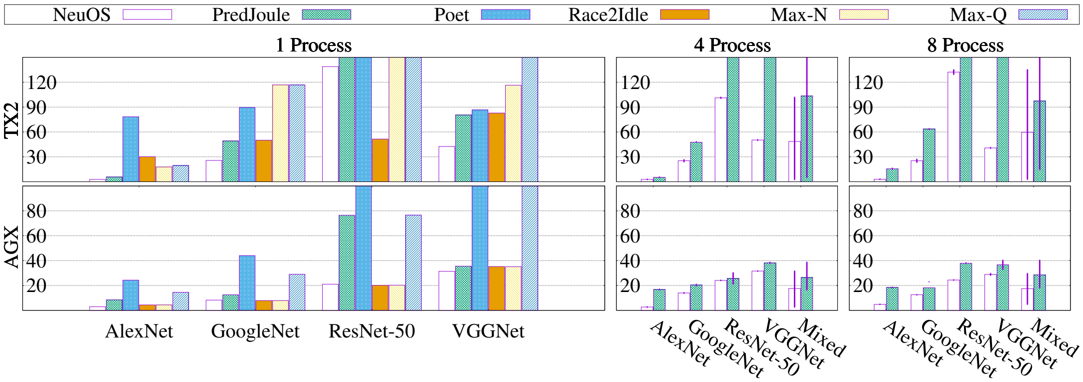


Figure 8: Latency under various methods (in ms) for 1, 4, and 8 instances of 4 DNN models.

5 Evaluation

In this section, we test our full implementation on top of Caffe [25] with an extensive set of evaluations.

5.1 Experimental Setup

In this section we lay out our experimental setup, which includes two embedded platforms and four popular DNN models. We compare NeuOS to 6 existing approaches.

Testbeds. We have chosen two different NVIDIA platforms imposing different architectural features (since deployed autonomous systems solutions, particularly for autonomous driving and robotics, seem to gravitate towards NVIDIA hardware as of writing this paper [26, 30]) to showcase the cross-platform nature of our design when it comes to hardware. We use NVIDIA Jetson TX2, with 6 big.LITTLE ARM-based cores and a 256-core Pascal based GPU with 11759 unique DVFS configurations, and the NVIDIA Jetson AGX Xavier, the latest powerful platform for robotics and autonomous vehicles with an 8-core NVIDIA Carmel CPU and a 512-core Volta-based GPU with 51967 unique DVFS configurations.

DNN models. Having a diversified portfolio of DNN models can showcase that NeuOS is future proof in the fast-moving field of neural networks. To that end, we use AlexNet [32], ResNet [19], GoogleNet [2], and VGGNet [49] in our experiments. Our method dynamically applies a lowrank version of a convolutional layer whenever approximation is necessary by keeping both version of the layer in memory for fast switching. The deadline for each DNN instance is based on their worst-case execution time (WCET) on each platform, and is set to 10ms, 30ms, 150ms, and 40ms respectively for Jetson TX2 and 5ms, 10ms, 25ms, 30ms respectively for AGX Xavier. Note that ResNet is much slower on Jetson TX2 due to the older JetPack software.

Small Cohort, Medium Cohort, Large Cohort sizes. We test NeuOS under three different cohort size classes to test for adaptability and balance: 1 process for small, 2 to 4 processes for medium, and 6 to 8 processes for large. Each of these cohort sizes have their own unique challenges. We measure average timing, energy consumption, and accuracy for these scenarios and provide a measure of balancing where applicable. For medium and large cohorts, we include a mixed scenario, where different DNN models are executed, which represents systems that use different DNNs (for example for voice and image recognition). For the medium cohort, one instance of each DNN model and for the large cohort, two instances of each model are initiated.

Adaptability to different system scenarios. As discussed in Sec. 4.1, we consider three different scenarios with different limits on latency, energy and accuracy: minimum energy, maximum accuracy, and balanced energy and accuracy.

Compared Solutions. We implement and compare six state-

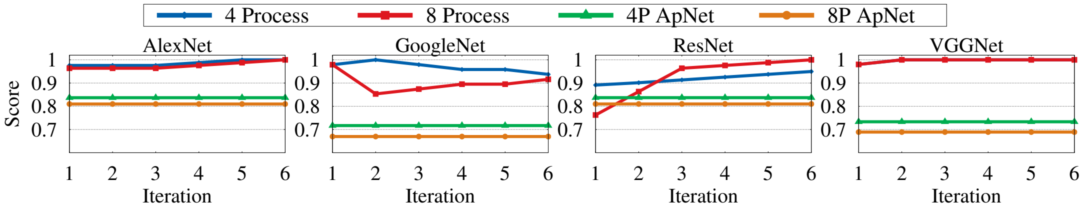


Figure 9: Average accuracy (y-axis as a fraction of 1) of the cohort over iteration of execution for 4 different DNN models with 4 and 8 instances compared to ApNet (x-axis is the iteration number).

of-the art solutions from the literature, including DNN-specific and DNN-agnostic ones, software-based DVFS and hardware-based DVFS, and application-level and system-level solutions. We present a short detail for each as follows.

PredJoule [5] is a system-level solution tailored towards DNN by employing a layer-based DVFS adjustment solution for optimization latency and energy. *Poet* [23] is a system-level control-theory based software solution that balances energy and timing in a best-effort manner via adjusting DVFS. We choose to compare against Poet instead of its extended approaches including JouleGuard [21] and CoAdapt [20], as they employ essentially the same set of control theory-based techniques as Poet. *ApNet* [3] is an application-level solution based on DNNs that can theoretically provide a per-layer approximation requirement offline to meet deadlines. *Race2Idle* [28] is the classic “run it as fast as you can” philosophy, which is always interesting to compare to. *Max-N* [10, 11] is a reactive hardware DVFS that maximizes frequency and sacrifices energy in the name of speed, in NVIDIA embedded hardware. *Max-Q* [10] is a hardware DVFS on Jetson TX2 that dynamically adjusts DVFS on the fly to conserve energy. However, this feature has been removed from the Xavier platform [11], and is replaced by low level power caps, such as 10W, 15W, and 30W. We use the 15w cap instead of Max-Q on Xavier.

5.2 Overall Effectiveness

In this section, we measure the efficacy of NeuOS on the two evaluated platforms under the balanced scenario. Since our design is concerned with timing predictability, energy consumption, and DNN accuracy, we measure all three constraints and compare against state-of-the-art literature under each platform and each scenario.

5.2.1 Small Cohort

Energy. The left column of Fig. 7 depicts our measurements in terms of average energy consumption compared to a GPU-enabled Poet, Max-Q, Max-N, PredJoule, and Race2Idle using AlexNet, GoogleNet, ResNet-50 and VGGNet as the base DNN model and using lowrank as the approximation method. As is evident in the figure, NeuOS is able to save energy considerably compared to all other methods on Jetson TX2 on all DNN models, with improvements of 68% on average for Jetson TX2 and 46% on average for AGX Xavier. This saving is due to the fact that in some cases accuracy is minimally traded off for the benefit of energy and timing.

On the Jetson AGX Xavier, NeuOS has better energy consumption compared to all other approaches on every DNN model except compared to PredJoule for VGGNet. As we shall see for timing, PredJoule misses the deadline of 30 for VGGNet, and NeuOS has decided to sacrifice energy to meet timing.

Latency. Fig. 8 shows the average execution time for NeuOS compared to the 5 methods and using 4 DNN models. NeuOS outperforms all other approaches, improving on average execution time by 68% on Jetson TX2 and by 40% on AGX Xavier. It is also interesting to note that AGX Xavier is much faster than Jetson TX2, by 70% on average.

Tail Latency. Through response time measurements, we find that NeuOS rarely misses the deadline (3.25% of the time). Moreover, the variance is low with the 99th percentile execution time for AlexNet, GoogleNet, ResNet-50, and VGGNet as 9.2 ms, 48 ms, 130.3 ms and 39.1 ms for TX2 and 5.0 ms, 12.0 ms, 26.1 ms and 36.2 ms for AGX respectively.

Accuracy. We also measure the accuracy loss of NeuOS compared to ground truth and compared to ApNet (Fig. 9 omits small cohort for clarity). ApNet is the only DNN-specific application level solution we are aware of. NeuOS has an approximation score of 0.94% on average (out of 1), which is better than ApNet by 21%.

5.2.2 Medium and Large Cohorts

In order to save space, we only compare PredJoule for the 4-process medium and 8-process large cohort sizes. In our testings, PredJoule already vastly outperforms other methodologies, and thus is a good comparison to NeuOS.

Energy. As is evident in the second and third columns of Fig. 7, NeuOS can almost always outperform PredJoule in terms of energy consumption on Jetson TX2 improving 70% on average. However, rather interestingly, NeuOS performs worse in terms of energy compared to PredJoule for GoogleNet, ResNet, and VGGNet on AGX Xavier. This is due to the fact that PredJoule again misses the deadlines on AGX Xavier, and NeuOS has sacrificed a negligible amount of energy (1.5% on average) in order to meet the deadline.

Latency. NeuOS always outperforms state-of the art, improving by 53% on average for Jetson TX2, and by 32% on average for AGX. This is due to the fact that NeuOS is able to leverage a small amount of accuracy and energy loss (in the case of AGX Xavier) for better timing and energy characteristics.

Tail Latency. We find that deadline miss ratio is about the

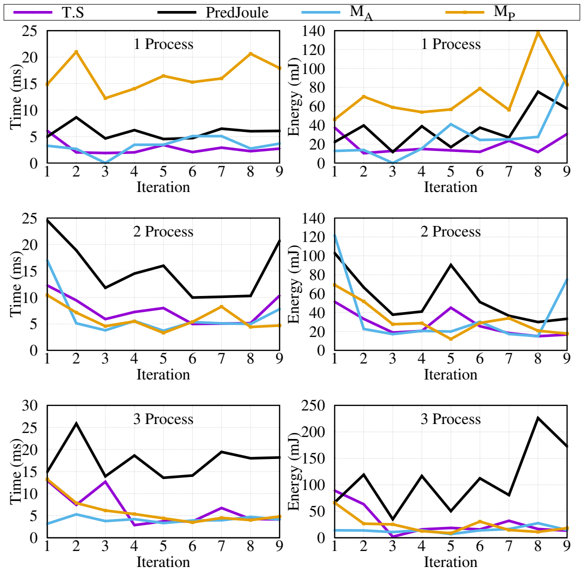


Figure 10: Performance of NeuOS under three scenarios compared to PredJoule on Jetson TX2.

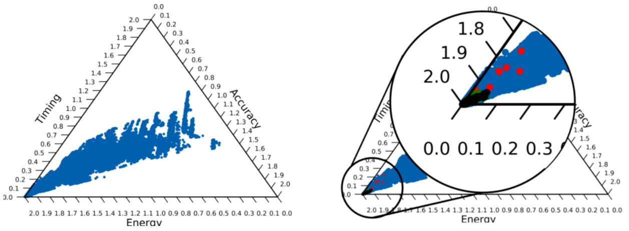
same as the small cohort. Moreover, the variance is similarly low with the 99th percentile execution time for AlexNet, GoogleNet, ResNet-50, and VGGNet as 10.4 ms, 39.2 ms, 101.7 ms and 69 ms for TX2 and 11 ms, 12.5, 26.3 and 35.9 ms for AGX respectively for the medium cohort and 13.6 ms, 40.8 ms, 190 ms and 72 ms for TX2 and 10.7 ms , 54 ms , 62 ms and 36.1 ms for AGX respectively for the large cohort.

Accuracy. Fig. 9 shows the average accuracy of the cohort over 6 iterations on AGX Xavier. As is evident in the figure, NeuOS generally improves upon accuracy as the system progresses because the optimization in Sec. 4.3 is able to find better DVFS configurations. When compared to the efficient approximation-aware solution APnet, NeuOS is able to achieve noticeably better accuracy in all scenarios.

Balance. A very important measure discussed in Sec. 3.3 is how balanced the system solution is when faced with multiple processes. To measure how balanced NeuOS is compared to PredJoule in the 4-process and 8-process scenarios, we include min-max bars in Fig. 7 and Fig. 8 to showcase the discrepancy between minimum and maximum timing/energy. As is evident in the figure, the discrepancy is negligible compared to the total energy consumption and execution time (up to 79 mJ and 4 ms). Thus, NeuOS maintains balance in the cohort. This is due to the coordinated cohorts and a uniform non-greedy decision making approach introduced in our design.

5.3 Detailed Examination on Tradeoff

In this section, we focus on the fact that system designers might require certain constraints that limits the ability of NeuOS in a certain dimension. To this end, we test our platform under three different scenarios: Maximum Accuracy (M_A), Minimum Power (M_P), and Balanced ($T.S$).



(a) The entire configuration space for all DVFS and accuracy combinations for Jetson TX2.

(b) Chosen configurations in the triangle space.

5.3.1 Energy and Latency.

We compare against PredJoule and measure average timing for the cohort in ms and average energy in mJ in Fig. 10 for 1-process (small), 2-process (medium), and 3-process (large) scenarios on AlexNet over 9 iterations on the Jetson TX2. PredJoule is shown as a black line. The deadline in this scenario is set to 25 to show the interesting characteristics of each method.

Balanced. As is evident in the figure, our statistical balanced approach outperforms PredJoule over all iterations. Notably in the case of medium and large cohorts, PredJoule has a particularly bad start in terms of timing and has higher fluctuation due to the greedy nature of DVFS selection.

Min Energy. Interestingly, M_P performs very bad (still meets the deadline) for the small cohort both in terms of timing and energy consumption. This is due to the algorithms discussed in Sec. 4.3. The system has switched to a very slow DVFS configuration to save energy. However, because of the non-linearity inherent in very slow DVFS configurations for GPUs [5], this has resulted in a very bad energy consumption as well. However, the coordination starts to pay off for medium and large cohorts. This is because a coordinated multi-process cohort needs faster DVFS configurations and thus, the circumstances push M_P out of the slow and power inefficient DVFS configuration subset. The greediness of PredJoule is inherent for medium and large cohorts in the form of very large fluctuations throughout the iterations.

Max Accuracy. M_A should improve accuracy while sacrificing energy and timing. We shall discuss the accuracy decisions shortly. However, the timing for M_A is worse than balanced energy by a negligible amount. The same is true for energy consumption (23% on average). This highlights a big design decision of the balanced scenario overall. Even for the balanced general approach, sacrificing accuracy is done very conservatively as was discussed earlier. Thus, the slight push toward perfect accuracy does not introduce very large overheads. However, as was discussed earlier guaranteeing a tight deadline (such as 10ms for AlexNet) requires some approximation if energy consumption is a consideration.

5.3.2 Energy-Accuracy Tradeoff.

For accuracy and to show where the variations of NeuOS jump in terms of system and application configurations, we bring back the triangle of Fig. 1, but with real DVFS and

Table 2: Average execution time overhead of NeuOS compared to other approaches on AlexNet (ms).

	1 Process	4 Process	8 Process
NeuOS	0.145	0.571	0.738
PredJoule	0.772	0.929	1.597
ApNet	0	3.27	5.85
Poet	151.03	604.12	1208.27

accuracy configurations with the selected configurations of M_P, M_A, and $T.S$ highlighted in Fig. 11b. As is evident in the triangle, the deadline limits the possible configurations to the bottom left corner. However, within that limitation, M_P (in red) has chosen configurations that are lower on energy consumption toward the upper right. On the other hand M_A (in green) has chosen configurations that are not as good in terms of energy consumption, but are better in terms of accuracy toward bottom left. Finally, $T.S$, colored black, is similar to M_A because of the high emphasis on accuracy in our design.

5.4 Overhead

Execution time: Table 2 shows the overhead of NeuOS compared to Poet, PredJoule, and ApNet using AlexNet as the baseline model on Jetson TX2 (times are in milliseconds). As is evident in Table 2, the overhead for NeuOS is negligible, especially compared to Poet and the overhead is also negligible compared to the overall execution time of AlexNet itself. The reason Poet is so slow is because it has to go through all DVFS configurations in a quadratic way ($O(n^2)$, n is the number of DVFS configurations). Even $O(n)$ would be unacceptable on embedded systems with more than 10000 unique DVFS configurations. This proves that applying our complexity reduction techniques (via hashing) is a must for a practical system solution. Moreover, NeuOS is more efficient than PredJoule and ApNet, especially in 4 Process and 8 Process scenarios.

Memory: As discussed in Sec. 5.1, our implementation keeps both the original and the lowrank approximation of the model in GPU memory for fast switching at layer boundaries. Moreover, NeuOS also holds the per-layer hash tables containing approximation and DVFS configuration information as described in Sec. 4. Table 3 depicts the added overhead in terms of both raw and percentage of the total NeuOS memory usage. As expected, the lowrank approximated version of each model has slightly less overall size compared to the original model. Nonetheless, this technique sacrifices memory overhead to improve latency and energy consumption. A viable alternative left as future work is dynamic approximation on the fly, which trades off latency for lower memory consumption. Moreover, the overhead of the hash tables is negligible compared to the total memory usage. Finally, the last two columns depict the cumulative maximum percentage of total available memory occupied by one instance of NeuOS for each platform (this memory usage also includes the temporary intermediate layer data [39]).

Table 3: Raw and percentage based memory overhead of applying NeuOS for each model.

	(a) Overhead in addition to Caffe			(b) Ratio to total memory	
	Lowrank	Hash Table	Ratio	Jetson TX2	AGX Xavier
AlexNet	226 MB	331 B	49%	10%	4%
GoogleNet	23 MB	2.1 KB	30%	2%	1%
ResNet-50	82 MB	3.2 KB	45%	7%	3%
VGGNet	509 MB	634 B	48%	25%	12.7%

6 Related Work

Trading off latency and power efficiency has been a hot topic in the related fields including real-time embedded systems and mobile computing [40, 8, 41, 53, 59, 16, 37, 42, 47, 1]. Due to the explosion of approximation techniques in different application domains, there has been several recent works [15, 35, 13] seeking to address this problem in a three dimensional space covering accuracy as well. Unfortunately, these works cannot resolve the problem as their applicability is limited in scope in various ways. JouleGuard [21], MeanTime [13], CoAdapt [20] and other similar approaches provide general system or hardware solution for non-DNN applications that claim to explore the three dimensional optimization space.

A very recent set of works including PredJoule [5], and ApNet [3] are able to provide latency predictability (meeting deadlines) for DNN-based workloads, yet they only consider two out of the three dimensions and focus on single-DNN scenarios. As we have discussed in Sec. 3.2, running ApNet and PredJoule at the same time even at different frequencies will result in a negative feedback loop. Thus, a better, more coordinated approach is required.

Such limitations would dramatically reduce the complexity of the optimization space, as the system-level and application-level tradeoffs focusing on a single task can be considered in an independent manner (e.g., pure system-level and application-level optimization under Poet [23] and CoAdapt, respectively). This results in solutions that are inapplicable to any practical autonomous real-time system featuring a multi-DNN environment.

7 Acknowledgment

We would like to thank our shepherd, Amitabha Roy, and the anonymous referees for their invaluable insight and advice into making this paper substantially better. This work is supported by NSF grant CNS CAREER 1750263.

8 Conclusion

This paper presents NeuOS, a comprehensive latency predictable system solution for running multi-DNN workloads in autonomous embedded systems. NeuOS can guarantee latency predictability, while managing energy optimization and dynamic accuracy adjustment based on specific system constraints via smart coordinated system- and application-level decision-making among multiple DNN instances. Extensive evaluation results prove the efficacy and practicality of NeuOS.

References

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 1–13. IEEE Press, 2016.

[2] Mohammadhossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab Mirrokni. Distributed balanced clustering via mapping coresets. In *Advances in Neural Information Processing Systems*, pages 2591–2599, 2014.

[3] Soroush Bateni and Cong Liu. Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 67–79, Dec 2018.

[4] Soroush Bateni and Cong Liu. Predictable data-driven resource management: an implementation using autoware on autonomous platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 339–352. IEEE, 2019.

[5] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 107–118, Dec 2018.

[6] Raunak P Bhattacharyya, Derek J Phillips, Changliu Liu, Jayesh K Gupta, Katherine Driggs-Campbell, and Mykel J Kochenderfer. Simulating emergent properties of human driving behavior using multi-agent reward augmented imitation learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 789–795. IEEE, 2019.

[7] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[8] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 27–39. IEEE Press, 2016.

[9] European Commission. Rolling plan for ict standardisation, 2018.

[10] NVIDIA Corp. Power management for jetson agx xavier devices. https://docs.nvidia.com/jetson/l4t/index.html#page/TegraLinuxDriverPackageDevelopmentGuide/power_management_jetson_xavier.html.

[11] NVIDIA Corp. Power management for jetson tx2 series devices. https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/TegraLinuxDriverPackageDevelopmentGuide/power_management_tx2_32.html.

[12] ETSI. Intelligent transport systems (its); vehicular communications; basic set of applications; part 3: Specifications of decentralized environmental notification basic service, Aug 2018.

[13] Anne Farrell and Henry Hoffmann. Meantime: Achieving both minimal energy and timeliness with approximate computing. In *USENIX Annual Technical Conference*, pages 421–435, 2016.

[14] ISO International Organization for Standardization. 26262: 2018. *Road vehicles—Functional safety*.

[15] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[17] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Henry Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 223–232, 2014.

[21] Henry Hoffmann. Jouleguard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 198–214. ACM, 2015.

[22] Sean Hollister. Tesla's new self-driving chip is here, and this is your best look yet., Apr 2019.

[23] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86, April 2015.

[24] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[25] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[26] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[27] Jaanus Kaugerand, Johannes Ehala, Leo Mõtus, and Jürgo-Sören Preden. Time-selective data fusion for in-network processing in ad hoc wireless sensor networks. *International Journal of Distributed Sensor Networks*, 14(11), 2018.

[28] D. H. K. Kim, C. Imes, and H. Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 78–85, Aug 2015.

[29] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[30] B. Kisacanin. Deep learning for autonomous vehicles. In *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 142–142, May 2017.

[31] Olga Kouchnarenko and Jean-François Weber. Adapting component-based systems at runtime via policies with temporal patterns. In *International Workshop on Formal Aspects of Component Software*, pages 234–253. Springer, 2013.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[33] TIMOTHY B. LEE. Tesla's autonomy event: Impressive progress with an unrealistic timeline, Apr 2019.

[34] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168. IEEE, 2011.

[35] Yongbo Li, Yurong Chen, Tian Lan, and Guru Venkataramani. Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1261–1270. IEEE, 2017.

[36] Shiyu Liang and R Srikant. Why deep neural networks for function approximation? *arXiv preprint arXiv:1610.04161*, 2016.

[37] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 255–266. IEEE Press, 2016.

[38] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[39] Zongqing Lu, Swati Rallapalli, Kevin Chan, and Thomas La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1663–1671, 2017.

[40] M. Maggio, E. Bini, G. Chasparis, and K. Årzén. A game-theoretic resource manager for rt applications. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 57–66, July 2013.

[41] Nikita Mishra, Huazhe Zhang, John D Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 267–281. ACM, 2015.

[42] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 267–278. IEEE Press, 2016.

[43] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[44] Francisca Rosique, Pedro J Navarro, Carlos Fernández, and Antonio Padilla. A systematic review of perception system and simulators for autonomous vehicles research. *Sensors*, 19(3):648, 2019.

[45] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6655–6659. IEEE, 2013.

[46] Rick Salay, Rodrigo Queiroz, and Krzysztof Czarnecki. An analysis of iso 26262: Using machine learning safely in automotive software. *arXiv preprint arXiv:1709.02435*, 2017.

[47] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[48] Akshay Kumar Shastry. *Functional Safety Assessment in Autonomous Vehicles*. PhD thesis, Virginia Tech, 2018.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[50] Nikolai Smolyanskiy, Alexey Kamenev, Jeffrey Smith, and Stan Birchfield. Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4241–4247. IEEE, 2017.

[51] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *17th IEEE Real-Time Systems Symposium*, pages 288–299, Dec 1996.

[52] Chen Tang, Zhuo Xu, and Masayoshi Tomizuka. Disturbance-observer-based tracking controller for neural network driving policy transfer. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[53] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[54] Jean-François Weber. Tool support for fuzz testing of component-based system adaptation policies. In *International Workshop on Formal Aspects of Component Software*, pages 231–237. Springer, 2016.

[55] Wikipedia contributors. Ternary plot - wikipedia,the free encyclopedia. `https://en.wikipedia.org/wiki/Ternary_plot`, 2019. [Online; accessed 31-May-2019].

[56] Jian Xue, Jinyu Li, Dong Yu, Mike Seltzer, and Yifan Gong. Singular value decomposition based low-footprint speaker adaptation and personalization for deep neural network. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 6359–6363. IEEE, 2014.

[57] Taro Yamane. Statistics: An introductory analysis. 1973.

[58] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint*, 2017.

[59] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *SIGPLAN Not.*, 51(4):545–559, March 2016.

[60] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic autonomous driving system testing. *arXiv preprint arXiv:1802.02295*, 2018.

PERCIVAL: Making In-Browser Perceptual Ad Blocking Practical with Deep Learning

Zainul Abi Din [†]
UC Davis

Panagiotis Tigas[†]
University of Oxford

Samuel T. King
UC Davis
Bouncer Technologies

Benjamin Livshits
Brave Software
Imperial College London

Abstract

In this paper we present PERCIVAL, a browser-embedded, lightweight, deep learning-powered ad blocker. PERCIVAL embeds itself within the browser's image rendering pipeline, which makes it possible to intercept every image obtained during page execution and to perform image classification based blocking to flag potential ads.

Our implementation inside both Chromium and Brave browsers shows only a minor rendering performance overhead of 4.55%, for Chromium, and 19.07%, for Brave browser, demonstrating the feasibility of deploying traditionally heavy models (i.e. deep neural networks) inside the critical path of the rendering engine of a browser. We show that our image-based ad blocker can replicate EasyList rules with an accuracy of 96.76%. Additionally, PERCIVAL does surprisingly well on ads in languages other than English and also performs well on blocking first-party Facebook ads, which have presented issues for rule-based ad blockers. PERCIVAL proves that image-based perceptual ad blocking is an attractive complement to today's dominant approach of block lists.

1 Introduction

Web advertising provides the financial incentives necessary to support most of the free content online, but it comes at a security and privacy cost. To make advertising effective, ad networks or publishers track user browsing behavior across multiple sites to generate elaborate user profiles for targeted advertising.

Users find that ads are intrusive [61] and cause disruptive browsing experience [6, 27]. In addition, studies have shown that advertisements impose privacy and performance costs to users, and carry the potential to be a malware delivery vector [2, 35, 37, 54, 55, 76].

Ad blocking is a software capability for filtering out unwanted advertisements to improve user experience, performance, security, and privacy. At present, ad blockers either run directly in the browser [4, 12] or as browser extensions [1].

Current ad blocking solutions filter undesired content based on "handcrafted" filter lists such as EasyList [74], which contain rules matching ad-carrying URLs and DOM elements. Most widely-used ad blockers, such as uBlock Origin [26] and Adblock Plus [1] use these block lists for content blocking. While useful, these approaches fail against adversaries who can change the ad-serving domain or obfuscate the web page code and metadata.

In an attempt to find a more flexible solution, researchers have proposed alternative approaches to ad blocking. One such approach is called Perceptual ad blocking, which relies on "visual cues" frequently associated with ads like the AdChoices logo or a sponsored content link. Storey et al. [70] built the first perceptual ad blocker that uses traditional computer vision techniques to detect ad-identifiers. Recently, Adblock Plus developers built filters into their ad blocker [15] to match images against a fixed template in order to detect ad labels. Due to the plethora of ad-disclosures, AdChoices logo and other ad-identifiers, it is unlikely that traditional computer vision techniques are sufficient and generalizable to the range of ads one is likely to see in the wild.

A natural extension to traditional vision-based blocking techniques is deep learning. Adblock Plus recently proposed SENTINEL [65] that detects ads in web pages using deep learning. SENTINEL's deep learning model takes as input the screenshot of the rendered webpage to detect ads. However, this technology is still in development.

To this end, we present PERCIVAL, a native, deep learning-powered *perceptual ad blocker*, which is built into the browser image rendering pipeline. PERCIVAL intercepts every image obtained during the execution sequence of a page and blocks images that it classifies as ads. PERCIVAL is small (half the average webpage size [25]) and fast, and we deploy it online within two commercial browsers to block and detect ads at real-time.

PERCIVAL can be run *in addition* to an existing ad blocker, as a last-step measure to block whatever slips through its

[†]Employed by Brave software when part of this work took place.

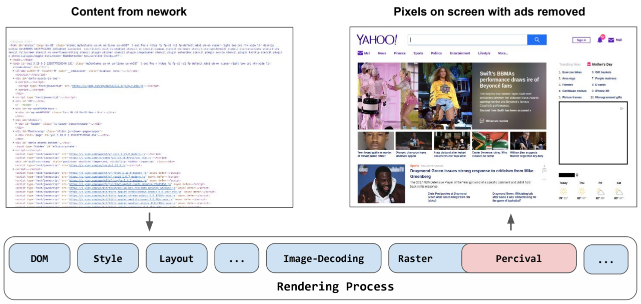


Figure 1: Overall architecture of PERCIVAL. PERCIVAL is positioned in the renderer process-which is responsible for creating rasterized pixels from HTML, CSS, Javascript. As the renderer process creates the DOM and decodes and rasterizes all image frames, these are first passed through PERCIVAL. PERCIVAL blocks the frames that are classified as ads. The corresponding output with ads removed is shown above (right).

filters. However, PERCIVAL may also be deployed *outside* the browser, for example, as part of a crawler, whose job is to construct comprehensive block lists to supplement EasyList.

1.1 Contributions

This paper makes the following contributions:

- **Perceptual ad blocking in Chromium-based browsers.** We deploy PERCIVAL in two Chromium-based browsers: Chromium and Brave. We demonstrate two deployment scenarios; first, PERCIVAL blocks ads synchronously as it renders the page, with a modest performance overhead. Second, PERCIVAL classifies images *asynchronously* and memoizes the results, thus speeding up the classification process[1].
- **Lightweight and accurate deep learning models.** We show that ad blocking can be done effectively using highly-optimized deep neural network-based models for image processing. Previous studies suggest that models over 5MB in size become hard to deploy on mobile devices [62]; because of our focus on low-latency detection, we create a compressed in-browser model that occupies 1.76MB[2] on disk, which is smaller by factor of 150 compared to other models of this kind [22], while maintaining similar accuracy results.
- **Accuracy and performance overhead measurements.** We show that our perceptual ad blocking model can replicate EasyList rules with the accuracy of 96.76%, making PERCIVAL a viable and complementary ad blocking layer. Our implementation within Chromium shows an average overhead of 178.23ms for page rendering. This overhead shows the feasibility of deploying deep neural networks inside the critical path of the rendering engine of the browser.
- **First-party ad blocking.** While the focus of traditional ad blocking is primarily on third-party ad blocking, we show that PERCIVAL blocks first-party ads as well, such as those found on Facebook. Specifically, our experiments show that PERCIVAL blocks ads on Facebook (often referred to as "sponsored content") with a 92% accuracy, with precision and recall of 78.4% and 70.0%.
- **Language-agnostic blocking.** We demonstrate that our model in PERCIVAL blocks images that are in languages we did not train our model on. We evaluate our trained model on Arabic, Chinese, Korean, French and Spanish image-based ads. Our model achieves an accuracy of 81.3% on Arabic, 95.1% on Spanish, and 93.9% on French datasets, with moderately high precision and recall. However, results from Chinese and Korean ads are less accurate.

2 Motivation

Intrusive, online, advertising has been a long standing concern for user privacy, security and overall web experience. While web advertising makes it easier and more economic for businesses to reach a wider audience, bad actors have exploited this channel to engage in malicious activities. Attackers use ad-distribution channels to hijack compromised web pages in order to trick users into downloading malware [54]. This is known as malicious advertising.

Mobile users are also becoming targets of malicious advertising [68]. Mobile applications contain code embedded from the ad networks, which provides the interface for the ad networks to serve ads. This capability has been abused by attackers where the landing page of the advertisements coming from ad networks links to malicious content. Moreover, intrusive advertisements significantly affect the user experience on mobile phones due to limited screen size [38]. Mobile ads also drain significant energy and network data [73].

Web advertising also has severe privacy implications for users. Advertisers use third party web-tracking by embedding code in the websites the users visit, to identify the same users again in a different domain, creating a more global view of the user browsing behavior [52]. Private user information is collected, stored and sold to other third party advertisers. These elaborate user profiles can be used to infer sensitive information about the users like medical history or political views [31,57]. Communication with these third party services is unencrypted, which can be exploited by attackers.

The security and privacy concerns surrounding web advertising has motivated research in ad blocking tools

[1] We make the source code, pre-trained models and data available for other researchers at https://github.com/dxaen/percival

[2] Our in-browser model is 3.2MB due to a less efficient serialization format. Still, the weights are identical to our 1.76MB model

from both academia [29, 40, 44, 69, 75] and industry notably Adblock Plus [1], Ghostery [13], Brave [4], Mozilla [47], Opera [16] and Apple [17]. Ad blocking serves to improve web security, privacy, usability, and performance. As of February 2017, 615 million devices had ad blockers installed [19] However, recently Google Chrome [14] and Safari [3] proposed changes in the API exposed to extensions, with the potential to block extension based ad-blockers. This motivates the need for native ad blockers like Brave [4], Opera [16], AdGraph [44], PageGraph [32] and even Percival.

3 Percival Overview

This paper presents Percival, a novel deep-learning based system for blocking ads. Our primary goal is to build a system that blocks ad images that could escape detection by current techniques, while remaining small and efficient enough to run in a mobile browser.

Figure 1 shows how Percival blocks rendering of ads. First, Percival runs in the browser image rendering pipeline. By running in the image rendering pipeline, Percival can inspect all images before the browser shows them to the user. Second, Percival uses a deep convolutional neural network (CNN) for detecting ad images. Using CNNs enables Percival to detect a wide range of ad images, even if they are in a language that Percival was not trained on.

This section discusses Percival's architecture overview, possible alternative implementations and detection model. Section 4 discusses the detailed design and implementation for our browser modifications and our detection model.

3.1 Percival's Architecture Overview

Percival's detection module runs in the browser's image decoding pipeline after the browser has decoded the image into pixels, but before it displays these pixels to the user. Running Percival after the browser has decoded an image takes advantage of the browser's mature, efficient, and extensive image decoding logic, while still running at a choke point before the browser displays the decoded pixels. Simply put, if a user sees an image, it goes through this pipeline first.

More concretely, as shown in Figure 1 Percival runs in the renderer process of the browser engine. The renderer process on receiving the content of the web page proceeds to create the intermediate data structures to represent the web page. These intermediate representations include the DOM-which encodes the hierarchical structure of the web page, the layout-tree, which consists of the layout information of all the elements of the web page, and the display list, which includes commands to draw the elements on the screen. If an element has an image contained within it, it needs to go through the *Image Decoding Step* before it can be rasterized. We run Percival after the *Image Decoding Step* during the *raster* phase which helps run Percival in parallel for multiple images at a time. Images that are classified as ads are blocked from rendering. The web page with ads removed is shown in Figure 1 (right). We present the detailed design and implementation in Section 4.

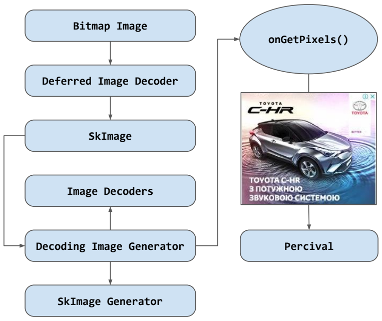


Figure 2: Percival in the image decoding pipeline. SkImage Generator allocates a bitmap and calls the `onGetPixels()` of `DecodingImageGenerator` to populate the bitmap. This bitmap is then passed to the network for classification and cleared if it contains an ad.

3.2 Alternative Possible Implementations and Advantages of Percival

One alternative to running Percival directly in the browser could have been to run Percival in the browser's JavaScript layer via an extension. However, this would require scanning the DOM to find image elements, waiting for them to finish loading, and then screenshotting the pixels to run the detection model. The advantage of a JavaScript-based system is that it works within current browser extensibility mechanisms, but recent work has shown how attackers can evade this style of detection [71].

Ad blockers that inspect web pages based on the DOM such as Ad Highlighter [70] are prone to DOM obfuscation attacks. They assume that the elements of the DOM strictly correspond to their visual representation. For instance, an ad blocker that retrieves all `img` tags and classifies the content contained in these elements does not consider the case, where a rendered image is a result of several CSS or JavaScript transformations and not the source contained in the tag. These ad blockers are also prone to resource exhaustion attacks where the publisher injects a lot of dummy elements in the DOM to overwhelm the ad blocker.

Additionally, a native implementation is much faster than a browser extension implementation with the added benefit of having access to the unmodified image buffers.

3.3 Detection Model

Percival runs a detection model on every image loaded in the document's main frame, a sub-document such as an `iframe`, as well as images loaded in JavaScript to determine if the image is an ad.

Although running directly within the browser provides Percival with more control over the image rendering process, it introduces a challenge: how to run the model efficiently in a browser? Our goal is to run PERCIVAL in browsers that run on laptops or even mobile phones. This requires that the model be small to be practical [62]. This design also requires that the model run directly in the image rendering pipeline, so overhead remains low. Any overhead adds latency to rendering for all images it inspects.

In PERCIVAL, we use the SqueezeNet [43] CNN as the starting point for our detection model. We modify the basic SqueezeNet network to be optimized for ad blocking by removing less important layers. This results in a model size that is less than 2MB and detects ad images in 11ms per image.

A second challenge in using small CNNs is how to provide enough training data. In general, smaller CNNs can have suitable performance but require more training data. What is more, the labels are highly imbalanced making the training procedure even more challenging.

Gathering ad images is non-trivial; most ads are programmatically inserted into the document through `iframe`s or JavaScript, and so simple crawling methods that work only on the initial HTML of the document will miss most of the ad images.

To crawl ad images, other researchers [22, 71] propose screenshotting iframes or JavaScript elements. This data collection method leads to problems with synchronizing the timing of the screenshot and when the element loads. Many screenshots end up with whites-space instead of the image content. Also, this method only makes sense if the input to the classifier is the rendered content of the web page.

To address these concerns and to provide ample training data, we design and implement a custom crawler in Blink[3] that handles dynamically-updated data and eliminates the race condition between the browser displaying the content and the screenshot we use to capture the image data. Our custom-crawler fetches ad and non-ad images directly from the rendering pipeline and uses the model trained during the previous phase as a labeler. This way we amplify our dataset to fine-tune our model further.

4 Design and Implementation of PERCIVAL

This section covers the design and implementation of the browser portion of PERCIVAL. We first cover the high-level design principles that guide our design, and then we discuss rendering and image handling in Blink, the rendering engine of Chromium-based browsers. Finally, we describe our end-to-end implementation within Blink.

4.1 Design Goals

We have two main goals in our design of PERCIVAL:

Run PERCIVAL at a choke point: Advertisers can serve ad images in different formats, such as JPG, PNG, or GIF. Depending on the format of the image, an encoded frame can traverse different paths in the rendering pipeline. Also, a wide range of web constructs can cause the browser to load images, including HTML image tags, JavaScript image objects, HTML Canvas elements, or CSS background attributes. Our goal is to find a single point in the browser to run PERCIVAL, such that it inspects all images, operates on pixels instead of encoded images, but does so before the user sees the pixels on the screen, enabling PERCIVAL to block ad images cleanly. **Note:** If individual pixels are drawn programmatically on canvas, PERCIVAL will not block it from rendering.

In Blink, the raster task within the rendering pipeline enables PERCIVAL to inspect, and potentially block, all images. Regardless of the image format or how the browser loads it, the raster task decodes the given image into raw pixels, which it then passes to the GPU to display the content on the screen. We run PERCIVAL at this precise point to abstract different image formats and loading techniques, while still retaining the opportunity to block an image before the user sees it.

Run multiple instances of PERCIVAL in parallel: Running PERCIVAL in parallel is a natural design choice because PERCIVAL makes all image classification decisions independently based solely on the pixels of each individual image. When designing PERCIVAL, we look for opportunities to exploit this natural parallelism to minimize the latency added due to the addition of our ad blocking model.

4.2 Rendering and PERCIVAL: Overview

We integrate PERCIVAL into Blink, the rendering engine for Google Chrome and Brave. From a high level, Blink's primary function is to turn a web page into the appropriate GPU calls [5] to show the user the rendered content.

A web page can be thought of as a collection of HTML, CSS, and JavaScript code, which the browser fetches from the network. The rendering engine parses this code to build the DOM and layout tree, and to issue OpenGL calls via `Skia`, Google's graphics library [24].

The layout tree contains the locations of the regions the DOM elements will occupy on the screen. This information together with the DOM element is encoded as a `display item`.

The browser then proceeds with rasterization, which takes the display items and turns them into bitmaps. Rasterization issues OpenGL draw calls via the Skia library to draw bitmaps. If the display list items have images in them (a common occurrence), the browser must decode these images before drawing them via Skia.

PERCIVAL intercepts the rendering process at this point, after the `Image Decode Task` and during the `Raster Task`. As the renderer process creates the DOM and decodes and rasterizes all image frames, these are first passed through

[3] Blink http://www.chromium.org/blink is the rendering engine used by Chromium.

Percival. Percival blocks the frames that are classified as ads.

4.3 End-to-End Implementation in Blink

We implement Percival inside Blink (Chromium rendering engine), where Percival uses the functionality exposed by the Skia library. Skia uses a set of image decoding operations to turn `SkImages`, which is the internal type within Skia that encapsulates images, into bitmaps. Percival reads these bitmaps and classifies their content accordingly. If Percival classifies the bitmap as an ad, it blocks it by removing its content. Otherwise, Percival lets it pass through to the next layers of the rendering process. When content is cleared, there are several ways to fill up the surrounding white-space; either collapsing it by propagating the information upwards or displaying a predefined image (user's spirit animal) in place of the ad.

Figure 2 shows an overview of our Blink integration. Blink class `BitmapImage` creates an instance of `DeferredImageDecoder` which in turn instantiates a `SkImage` object for each encoded image. SkImage creates an instance of `DecodingImageGenerator` (blink class) which will in turn decode the image using the relevant image decoder from Blink. Note that the image hasn't been decoded yet since chromium practices deferred image decoding.

Finally, `SkImageGenerator` allocates bitmaps corresponding to the encoded `SkImage`, and calls `onGetPixels()` of `DecodingImageGenerator` to decode the image data using the proper image decoder. This method populates the buffer (pixels) that contain decoded pixels, which we pass to Percival along with the image height, width, channels information (`SKImageInfo`) and other image metadata. Percival reads the image, scales it to 224 × 224 × 4 (default input size expected by SqueezeNet), creates a tensor, and passes it through the CNN. If Percival determines that the buffer contains an ad, it clears the buffer, effectively blocking the image frame.

Rasterization, image decoding, and the rest of the processing happen on a raster thread. Blink rasters on a per tile basis and each tile is like a resource that can be used by the GPU. In a typical scenario there are multiple raster threads each rasterizing different raster tasks in parallel. Percival runs in each of these worker threads after image decoding and during rasterization, which runs the model in parallel.

As opposed to Sentinel [65] and Ad Highlighter [36] the input to Percival is not the rendered version of web content; Percival takes in the Image pixels directly from the image decoding pipeline. This is important since with Percival we have access to unmodified image buffers and it helps prevent attacks where publishers modify content of the webpage (including iframes) with overlaid masks (using CSS techniques) meant to fool the ad blocker classifier.

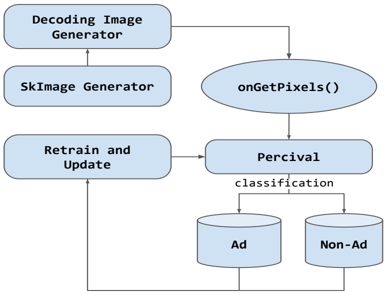


Figure 3: Crawling, labelling and re-training with Percival. Every decoded image frame is passed through Percival and Percival downloads the image frame into the appropriate bucket.

5 Deep Learning Pipeline

This section covers the design of Percival's deep neural network and the corresponding training workflow. We first describe the network employed by Percival and the training process. We then describe our data acquisition and labelling techniques.

5.1 Percival's CNN Architecture

We cast ad detection as a traditional image classification problem, where we feed images into our model and it classifies them as either being (1) an ad, or (2) not an ad. CNNs are the current standard in the computer vision community for classifying images.

Because of the prohibitive size and speed of standard CNN based image classifiers, we use a small network, SqueezeNet [43], as the starting point for our in-browser model. The SqueezeNet authors show that SqueezeNet achieves comparable accuracy to much larger CNNs, like AlexNet [48], and boasts a final model size of 4.8 MB.

SqueezeNet consists of multiple *fire modules*. A *fire module* consists of a "squeeze" layer, which is a convolution layer with 1 × 1 filters and two "expand" convolution layers with filter sizes of 1 × 1 and 3 × 3, respectively. Overall, the ”squeeze” layer reduces the number of input channels to larger convolution filters in the pipeline.

A visual summary of Percival's network structure is shown in Figure 4. As opposed to the original SqueezeNet, we down-sample the feature maps at regular intervals in the network. This helps reduce the classification time per image. We also perform max-pooling after the first convolution layer and after every two fire modules.

5.2 Data Acquisition

We use two systems to collect training image data. First, we use a traditional crawler with traditional ad-blocking rules (EasyList [7]) to identify ad images. Second, we use our browser instrumentation from Percival to collect images, improving on some of the issues we encountered with our traditional crawler.

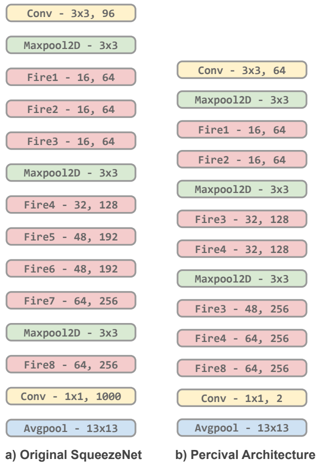


Figure 4: Original SqueezeNet (left) and PERCIVAL's fork of SqueezeNet (right). For `Conv`, `Maxpool2D`, and `Avgpool` blocks $a \times b$ represents the dimensions of the filters used. For fire blocks *a, b* represents the number of intermediate and output channels. We remove extraneous blocks as well as downsample the feature maps at regular intervals to reduce the classification time per image.

5.2.1 Crawling with EasyList

We use a traditional crawler matched with a traditional rule-based ad blocker to identify ad content for our first dataset. In particular, to identify ad elements which could be iframes or complex JavaScript constructs, we use EasyList, which is a set of rules that identify ads based on the URL of the elements, location within the page, origin, class or id tag, and other hand-crafted characteristics known to indicate the presence of ad content.

We built a crawler using Selenium [21] for browser automation. We then use the crawler to visit Alexa top-1,000 web sites, waiting for 5 seconds on each page, and then randomly selecting 3 links and visiting them, while waiting on each page for a period of 5 seconds as before. For every visit, the crawler applies every EasyList network, CSS and exception rule.

For every element that matches an EasyList rule, our crawler takes a screenshot of the component, cropped tightly to the coordinates reported by Chromium, and then stores it as an ad sample. We capture non-ad samples by taking screenshots of the elements that do *not* match any of the EasyList rules. Using this approach we, extract 22,670 images out of which 13,741 are labelled as ads, and 8,929 as non-ads. This automatic process was followed by a semi-automated post-processing step, which includes removing duplicate images, as well as manual spot-checking for misclassified images.

Eventually, we identify 2,003 ad images and 7,432 non-ad images. The drop in the number of ad images from 13,741 to 2,003 is due to a lot duplicates and content-less (single-color) images due to the asynchrony of iframe-loading and the timing of the screenshot. These shortcomings motivated our new crawler. To balance the positive and negative examples in our dataset so the classifier doesn't favor one class over another, we limited the number of non ad and ad images to 2,000.

5.2.2 Crawling with PERCIVAL

We found that traditional crawling was good enough to bootstrap the ad classification training process, but it has the fundamental disadvantage that for dynamically-updated elements, the meaningful content is often unavailable at the time of the screenshot, leading to screenshots filled with white-space.

More concretely, the *page load* event is not very reliable when it comes to loading iframes. Oftentimes when we take a screenshot of the webpage after the *page load* event, most of the iframes do not appear in the screenshots. Even if we wait a fixed amount of time before taking the screenshot, iframes constantly keep on refreshing, making it difficult to capture the rendered content within the iframe consistently.

To handle dynamically-updated data, we use PERCIVAL's browser architecture to read all image frames after the browser has decoded them, eliminating the race condition between the browser displaying the content and the screenshot we use to capture the image data. This way we are guaranteed to capture all the iframes that were rendered, independently of the time of rendering or refresh rate.

Instrumentation: Figure 3 shows how we use PERCIVAL's browser instrumentation to capture image data. Each encoded image invokes an instance of `DecodingImageGenerator` inside Blink, which in turn decodes the image using the relevant image decoder (PNG, GIFs, JPG, etc.). We use the buffer passed to the decoder to store pixels in a bitmap image file, which contains exactly what the rendering engine sees. Additionally, the browser passes this decoded image to PERCIVAL, which determines whether the image contains an ad. This way, every time the browser renders an image, we automatically store it and label it using our initially trained network, resulting in a much cleaner dataset.

Crawling: To crawl for ad and non-ad images, we run our PERCIVAL-based crawler with a browser automation tool called Puppeteer [20]. In each phase, the crawler visits the landing page of each Alexa top-1,000 websites, waits until `networkidle0` (when there are no more than 0 network connections for at least 500 ms) or 60 seconds. We do this to ensure that we give the ads enough time to load. Then our crawler finds all internal links embedded in the page. Afterwards, it visits 20 randomly selected links for each page, while waiting for `networkidle0` event or 60 seconds time out on each request.

In each phase, we crawl between 40,000 to 60,000 ad images. We then post process the images to remove duplicates, leaving around 15-20% of the collected results as useful. We

Images	Ads Identified	Accuracy	Precision	Recall
6,930	3466	96.76%	97.76%	95.72%

Figure 5: Summary of the results obtained by testing the dataset gathered using EasyList with PERCIVAL.

crawl for a total of 8 phases, retraining PERCIVAL after each stage with the data obtained from the current and all the previous crawls. As before, we cap the number of non-ad images to the amount of ad images to ensure a balanced dataset.

This process was spread-out in time over 4 months, repeated every 15 days for a total of 8 phases, where each phase took 5 days. Our final dataset contains 63,000 unique images in total with a balanced split between positive and negative samples.

6 Evaluation

6.1 Accuracy Against EasyList

To evaluate whether PERCIVAL can be a viable shield against ads, we conduct a comparison against the most popular crowd-sourced ad blocking list, EasyList [7], currently being used by extensions such as Adblock Plus [1], uBlock Origin [26] and Ghostery [13].

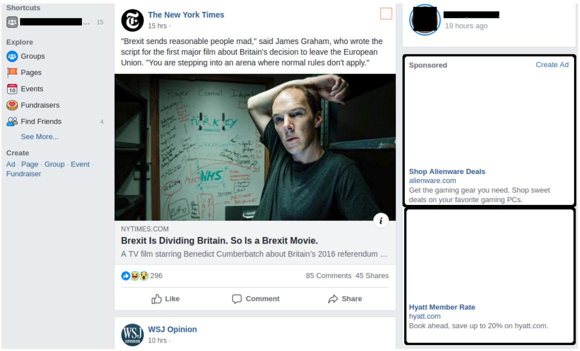
Methodology: For this experiment, we crawl Alexa top 500 news websites as opposed to Alexa top 1000 websites used in the crawl for training. This is because news websites are an excellent source of advertisements [18] and the crawl can be completed relatively quickly. Also, Alexa top 500 news websites serves as a test domain different from the train domain we used previously.

For our comparison we create two data sets: First, we apply EasyList rules to select DOM elements that potentially contain ads (IFRAMEs, DIVs, etc.); we then capture screenshots of the contents of these elements. Second, we use resource-blocking rules from EasyList to label all the images of each page according to their resource URL. After crawling, we manually label the images to identify the false positives resulting in a total of 6,930 images.

Performance: On our evaluation dataset, PERCIVAL is able to replicate the EasyList rules with accuracy 96.76%, precision 97.76% and recall 95.72% (Figure 5), illustrating a viable alternative to the manually-curated filter-lists.

Ads	No-ads	Accuracy	FP	FN	Precision	Recall
354	1,830	92.0%	68	106	78.4%	70.0%

Figure 6: Online evaluation of Facebook ads and sponsored content.



The New York Times
15 hrs ·

"Brexit sends reasonable people mad," said James Graham, who wrote the script for the first major film about Britain's decision to leave the European Union. "You are stepping into an arena where normal rules don't apply."

NYTIMES.COM
Brexit Is Dividing Britain. So Is a Brexit Movie.
A TV film starring Benedict Cumberbatch about Britain's 2016 referendum ...

WSJ Opinion
10 hrs ·

[redacted]
19 hours ago

Shop Alienware Deals
alienware.com
Get the gaming gear you need. Shop sweet deals on your favorite gaming PCs.

Hyatt Member Rate
hyatt.com
Book ahead, save up to 20% on hyatt.com.

Figure 7: The screenshots show one of the author' Facebook home page accessed with PERCIVAL. The black rectangles are not part of the original screenshot.

6.2 Blocking Facebook Ads

Facebook obfuscates the "signatures" of ad elements (e.g. HTML classes and identifiers) used by filter lists to block ads since its business model depends on serving first-party ads. As of now, Facebook does not obfuscate the content of sponsored posts and ads due to the regulations regarding misleading advertising [10, 11]. Even though this requirement favors perceptual ad blockers over traditional ones, a lot of the content on Facebook is user-created which complicates the ability to model ad and non-ad content.

In this section, we assess the accuracy of PERCIVAL on blocking Facebook ads and sponsored content.

Methodology: To evaluate PERCIVAL's performance on Facebook, we browse Facebook with PERCIVAL for a period of 35 days using two non-burner accounts that have been in use for over 9 years. Every visit is a typical Facebook browsing session, where we browse through the feed, visit friends' profiles, and different pages of interest. For desktop computers two most popular places to serve ads is the right-side columns and within the feed (labelled sponsored) [9].

For our purposes, we consider content served in these elements as ad content and everything else as non-ad content. A false positive (FP) is defined as the number of non-ads incorrectly blocked and false negative (FN) is the number of ads PERCIVAL missed to block. For every session, we manually compute these numbers. Figure 6 shows the aggregate numbers from all the browsing sessions undertaken. Figure 7 shows PERCIVAL blocking right-side columns correctly.

Results: Our experiments show that PERCIVAL blocks ads on Facebook with a 92% accuracy and 78.4% and 70.0% as precision and recall, respectively. Figure 6 shows the complete results from this experiment. Even though we achieve the accuracy of 92%, there is a considerable number of false positives and false negatives, and as such, precision and recall are lower. The classifier always picks out the ads in the right-columns but struggles with the ads embedded in the feed. This is the source of majority of the false negatives. False positives come from high "ad intent" user-created content, as well as content created by brand or product pages on Facebook (Figure 8).

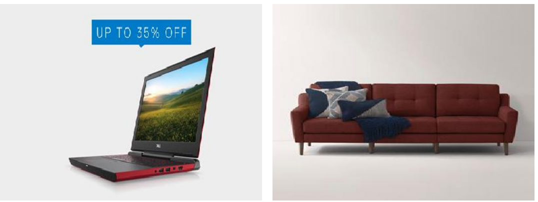


Figure 8: Examples of false positives and false negatives on Facebook (left) **False Positive:** This post was created by page owned by Dell Corp. (right) **False Negative:** This post was part of the sponsored content in the news feed.

Discussion: False Positives and False Negatives: To put Figure 6 into perspective since it might appear to have an alarming number of false positives and false negatives, it is worthwhile to consider an average scenario. If each facebook visit on average consists of browsing through 100 images, then by our experiments, a user will find roughly 16 ad images and 84 non-ad images, out of which PERCIVAL will block 11 to 12 ad images on average while also blocking 3 to 4 non-ad images. This is shown in Figure 10.

In addition to the above mentioned experiments which evaluate the out of box results of using PERCIVAL, we trained a version of PERCIVAL on a particular user's ad images. The model achieved higher precision and recall of 97.25%, 88.05% respectively.

6.3 Blocking Google Image Search Results

To improve our understanding of the misclassifications of PERCIVAL, we used Google Images as a way to fetch images from distributions that have high or low ad intent. For example, we fetched results with the query "Advertisement" and used PERCIVAL to classify and block images. As we can see in Figure 11, out of the top 23 images, 20 of them were successfully blocked. Additionally, we tested with examples of low ad intent distribution we used the query "Obama"). We also searched for other keywords, such as "Coffee", "Detergent", etc. The detailed results are presented in Figure 12. As shown, PERCIVAL can identify a significant percentage of images on a highly ad-biased content.

6.4 Language-Agnostic Detection

We test PERCIVAL against images with language content different than the one we trained on. In particular, we source a data set of images in Arabic, Chinese, French, Korean and Spanish.

Language	# crawled	# Ads	Accuracy	Precision	Recall
Arabic	5008	2747	81.3%	83.3%	82.5%
Spanish	2539	309	95.1%	76.8%	88.9%
French	2414	366	93.9%	77.6%	90.4%
Korean	4296	506	76.9%	54.0%	92.0%
Chinese	2094	527	80.4%	74.2%	71.5%

Figure 9: Accuracy of PERCIVAL on ads in non-English languages. The second column represents the number of images we crawled, while the third column is the number of images that were identified as ads by a native speaker. The remaining columns indicate how well PERCIVAL is able to reproduce these labels.

Images	Ads	No-ads	FP	FN
100	16	84	3-4	4-5

Figure 10: Average reporting of evaluation of Facebook ads and sponsored content per visit. We assume each Facebook visit consists of browsing through 100 total images.

Crawling: To crawl for ad and non-ad images, we use ExpressVPN [8] to VPN into major world cities where the above mentioned languages are spoken. For instance, to crawl Korean ads, we VPN into two locations in Seoul. We then manually visit top 10 websites as mentioned in SimilarWeb [23] list. We engage with the ad-networks by clicking on ads, as well as closing the ads (icon at the top right corner of the ad) and then choosing random responses like content not relevant or ad seen multiple times. This is done to ensure we are served ads from the language of the region.

We then run PERCIVAL-based crawler with the browser automation tool Puppeteer [20]. Our crawler visits the landing page of each top 50 SimilarWeb websites for the given region, waits until `networkidle0` (when there are no more than 0 network connections for at least 500 ms) or 60 seconds. Then our crawler finds all internal links embedded in the page. Afterwards, it visits 10 randomly selected links for each page, while waiting for `networkidle0` event or 60 seconds time out

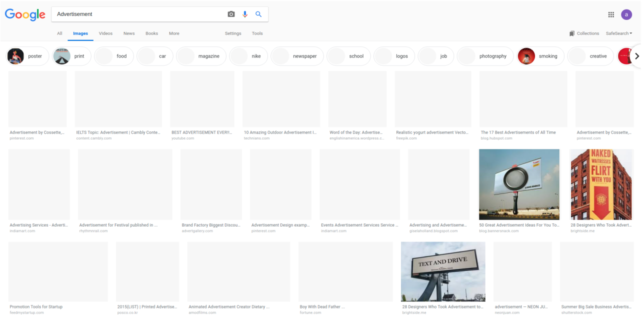


Figure 11: Search results from searching for "Advertisement" on Google images, using PERCIVAL.

Query	# blocked	# rendered	FP	FN
Obama	12	88	12	0
Advertisement	96	4	0	4
Coffee	23	77	-	-
Detergent	85	15	10	6
iPhone	76	24	23	1

Figure 12: PERCIVAL blocking image search results. For each search we only consider the first 100 images returned ("-" represents cases where we were not able to determine whether the content served is ad or non-ad).

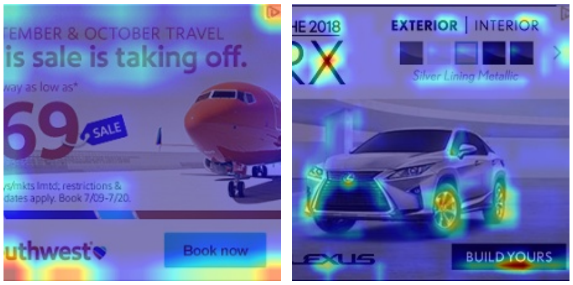
on each request. As opposed to Section 5.2.2, we download every image frame to a single bucket.

Labeling: For each language, we crawl 2,000–6,000 images. We then hire a native speaker of the language under consideration and have them label the data crawled for that language. Afterwards, we test PERCIVAL with this labeled dataset to determine how accurately can PERCIVAL reproduce these human annotated labels. Figure 9 shows the detailed results from all languages we test on. Figure 14 shows a screen shot of a Portuguese website rendered with PERCIVAL.

Results: Our experiments show that PERCIVAL can generalize to different languages with high accuracy (81.3% for Portuguese, 95.1% for Spanish, 93.9% for French) and moderately high precision and recall (83.3%, 82.5% for Arabic, 76.8%, 88.9% for Spanish, 77.6%, 90.4% for French). This illustrates the out-of-the box benefit of using PERCIVAL for languages that have much lower coverage of EasyList rules, compared to the English ones. The model does not perform as well on Korean and Chinese datasets.

6.5 Salience Map of the CNN

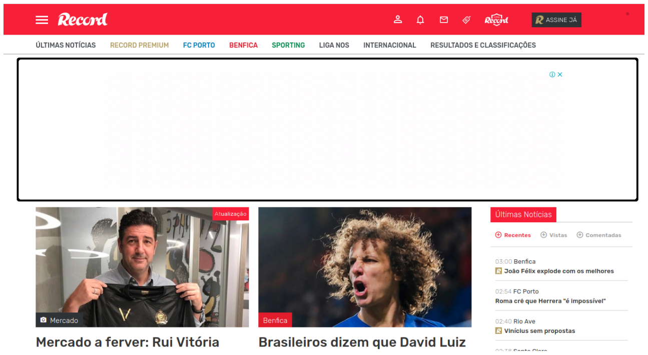
To visualize which segments of the image are influencing the classification decision, we used Grad-CAM [64] network salience mapping which allow us to highlight the important regions in the image that caused the prediction. As we can see in Figure 13, our network is focusing on ad visual cues (*AdChoice* logo), when this is present (case (a)), also it follows the outlines of text (signifying existence of text between white space) or identifies features of the object of interest (wheels of a car).



(a) Ad image: Layer 9

(b) Ad image: Layer 5

Figure 13: Salience map of the network on a sample ad images. Each image corresponds to the output of Grad-CAM [64] for the layer in question.



Mercado a ferver: Rui Vitória

Brasileiros dizem que David Luiz

03:00 Benfica
João Félix explode com os melhores

02:54 FC Porto
Roma crê que Herrera "é impossível"

02:40 Rio Ave
Vinícius sem propostas

Figure 14: PERCIVAL results on record.pt (Portuguese language website).

6.6 Runtime Performance Evaluation

We next evaluate the impact of PERCIVAL-based blocking on the browser performance. This latency is a function to the number and complexity of the images on the page and the time the classifier takes to classify each of them. We measure the rendering time impact when we classify each image *synchronously*.

To evaluate the performance of our system, we used top 5,000 URLs from Alexa to test against Chromium compiled on Ubuntu Linux 16.04, with and without PERCIVAL activated. We also tested PERCIVAL in Brave, a privacy-oriented Chromium-based browser, which blocks ads using block lists by default. For each experiment we measured `render` time which is defined as the difference between `domComplete` and `domLoading` events timestamps. We conducted the evaluations sequentially on the same Amazon m5.large EC2 instance to avoid interference with other processes and make the comparison fair. Also, all the experiments were using `xvfb` for rendering, an in-memory display server which allowed us to run the tests without a display.

In our evaluation we show an increase of 178.23ms of median render time when running PERCIVAL in the rendering critical path of Chromium and 281.85ms when running inside Brave browser with ad blocker and shields on. Figures 15 and 16 summarize the results.

To capture rendering and perceptual impact better, we create a micro-benchmark with `firstMeaningfulPaint` to illustrate overhead. In our new experiment, we construct a static html page containing 100 images. We then measure `firstMeaningfulPaint` with Percival classifying images synchronously and asynchronously. In synchronous classification, PERCIVAL adds 120ms to Chrome and 140ms

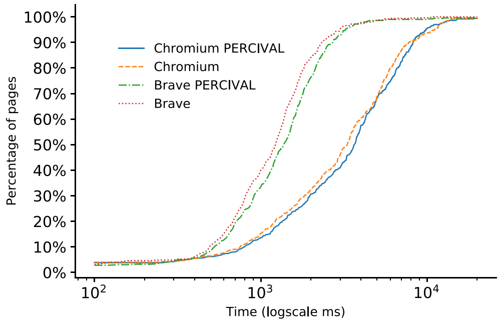


Figure 15: `Render` time evaluation in Chromium and Brave browser.

to Brave. In asynchronous classification, PERCIVAL adds 6ms to Chrome and 3ms to Brave. Although asynchronous classification nearly eliminates overhead, it opens up the possibility of showing an image to the user that we later remove after flagging it as an ad because the rasterization of the image runs in parallel with classification in this mode of operation.

To determine why PERCIVAL with Brave is slower than Chromium. We trace events inside the decoding process using `firstMeaningfulPaint` and confirm there is no significant deviation between the two browsers. The variance observed initially is due to the additional layers in place like Brave's ad blocking shields.

6.7 Comparison With Other Deep Learning Based Ad Blockers

Recently, researchers evaluated the accuracy of three deep-learning based perceptual ad blockers including PERCIVAL [71]. They used real website data from Alexa top 10 news websites to collect data which is later manually labelled. In this evaluation, PERCIVAL outperformed models 150 times bigger than PERCIVAL in terms of recall. We show their results in Figure 17.

6.8 Adversarial Attacks against PERCIVAL

In recent work by Tramèr et al. [71], they show how the implementation of some state-of-the-art perceptual ad blockers, including PERCIVAL, is vulnerable to attacks.

First, the authors in [71] claim that one adversarial sample influences PERCIVAL to block another benign non-ad image. This, however, is not true; the authors claim to use two benign images, one of which is not benign and other is contentless white-space image. PERCIVAL blocks these images. If these are replaced with stock non-ad images, PERCIVAL correctly renders both, meaning that PERCIVAL makes each decision independently and is not vulnerable to hijacking as is claimed in the paper.

Baseline	Treatment	Overhead (%)	(ms)
Chromium	Chromium + PERCIVAL	4.55	178.23
Brave	Brave + PERCIVAL	19.07	281.85

Figure 16: Performance evaluation of PERCIVAL on `Render` metric.

We found that one of the attacks where they used PERCIVAL's model to create adversarial ad images affects PERCIVAL due to our design decision to run PERCIVAL client-side thereby giving attackers white box access to the model. To address this concern, we argue that PERCIVAL is extremely light-weight and can be re-trained and updated very quickly. Our model currently takes 9 minutes (7 epochs) to fine-tune the weights of the network on an NVIDIA V 100 GPU, meaning that we can generate new models very quickly. PERCIVAL is 1.7MB which is almost half the average web page in 2018 [25] making frequent downloads easier.

To demonstrate, re-training and model update as an effective defense against the adversarial samples, we trained a MobilenetV2 [63] with our current dataset. It took 9 minutes of fine-tuning to get to our baseline accuracy. The updated model correctly classified all the adversarial samples generated for PERCIVAL by Tramer et al. [71] suggesting that none of the samples transferred to this model. It should be noted that, we did not add any more data to our dataset.

While we do accept that given sufficient time and machine learning expertise, it may be possible to create adversarial samples that generalize across different models but it in effect makes evasion more expensive. If we can update the model frequently, adversaries will have to play catch-up every time.

Additionally, to improve the robustness of the models against adversarial attacks one could employ techniques like *min-max* (robust) optimization [56] ,where the classification loss is minimized while maximizing the acceptable perturbation one can apply to the image, or *randomized smoothing* [34, 51, 53] where provable (or certified) robust accuracy can be afforded. Such techniques have shown promising results in training robust models and are currently under active research [66, 78].

Two main criticisms with such techniques is the performance degradation in accuracy but also the costly optimization involved. Although the "inherent tension" between robustness and accuracy [72] is inevitable, the l_2 perturbations drive the network to focus on more perceptual features and not on imperceptual features that can be exploitable. The training time penalty though can be mitigated by adopting fast *min-max*-based adversarial robustness training algorithms like [67, 79]. Given the fast iteration time for fine-tuning our network, any such performance degradation should be within our iteration cycle quota. We leave thorough study of such mitigation techniques for future work.

Model	Size	FP	FN
Sentinel [22] Clone	256 MB	0/20	5/29
ResNet [42]	242 MB	0/20	21/39
PERCIVAL	1.76 MB	2/7	3/33

Figure 17: Tramer et al.'s [71] evaluation of various deep learning based perceptual ad blockers. The difference in the number of images used for evaluation stem from the kind of images the ad blocker is expecting.

7 Limitations

Dangling Text: By testing PERCIVAL integrated into Chromium, we noticed the following limitations. Many ads consist of multiple elements, which contain images and text information layered together. PERCIVAL is positioned in the rendering engine, and therefore it has access to one image at a time. This leads to situations where we effectively block the image, but the text is left dangling. Although this is rare, we can mitigate this by retraining the model with ad image frames containing just the text. Alternatively, a non-machine learning solution would be to memorize the DOM element that contains the blocked image and filter it out on consecutive page visitations. Although this might provide an unsatisfying experience to the user, we argue that it is of the benefit of the user to eventually have a good ad blocking experience, even if this is happening on a second page visit.

Small Images: Currently, images that are below 100×100 size skips PERCIVAL to reduce the processing time. This is a limitation which can be alleviated by deferring the classification and blocking of small images to a different thread, effectively blocking *asynchronously*. That way we make sure that we don't regress the performance significantly, while we make sure that consecutive requests will continue blocking small ads.

8 Related Work

Filter lists: Popular ad blockers like, Adblock Plus [1], uBlock Origin [26], and Ghostery [13] are using a set of rules, called filter-list, to block resources that match a predefined crowd-sourced list of regular expressions (from lists like EasyList and EasyPrivacy). On top of that, CSS rules are applied, to prevent DOM elements that are potential containers of ads. These filter-lists are crowd-sourced and updated frequently to adjust on the non-stationary nature of the online ads [70]. For example, EasyList, the most popular filter-list, has a history of 9 years and contains more than 60,000 rules [74]. However, filter-list based solutions enable a continuous cat-and-mouse game: their maintenance cannot scale efficiently, as they depend on the human-annotator and they do not generalize to "unseen" examples.

Perceptual Ad Blocking: Perceptual ad blocking is the idea of blocking ads based solely on their appearance; an example ad, highlighting some of the typical components. Storey et al. [70] uses the rendered image content to identify ads. More specifically, they use OCR and fuzzy image search techniques to identify *visual cues* such as ad disclosure markers or sponsored content links. Unlike PERCIVAL, this work assumes that the ad provider is complying with the legislation and is using visual cues like *AdChoices*.

Sentinel [65] proposes a solution based on convolutional neural networks (CNNs) to identify Facebook ads. This work is closer to our proposal; however, their model is not deployable in mobile devices or desktop computers because of its large size (>200MB). Also, we would like to mention the work of [28, 42, 77], where they use deep neural networks to identify the represented signifiers in the Ad images. This is a promising direction in semantic and perceptual ad blocking.

Adversarial attacks: In computer-vision, researchers have demonstrated attacks that can cause prediction errors by near-imperceptible perturbations of the input image. This poses risks in a wide range of applications on which computer vision is a critical component (e.g. autonomous cars, surveillance systems) [58–60]. Similar attacks have been demonstrated in speech to text [30], malware detection [39] and reinforcement-learning [41]. To defend from adversarial attacks, a portfolio of techniques has been proposed [33, 45, 46, 49, 50, 56], whether these solve this open research problem, remains to be seen.

9 Conclusion

With PERCIVAL, we illustrate that it is possible to devise models that block ads, while rendering images inside the browser. Our implementation shows a rendering time overhead of 4.55%, for Chromium and and 19.07%, for Brave browser, demonstrating the feasibility of deploying deep neural networks inside the critical path of the rendering engine of a browser. We show that our perceptual ad blocking model can replicate EasyList rules with an accuracy of 96.76%, making PERCIVAL a viable and complementary ad blocking layer. Finally, we demonstrate off the shelf language-agnostic detection due to the fact that our models do not depend on textual information and we show that PERCIVAL is a compelling blocking mechanism for first-party Facebook sponsored content, for which traditional filter based solutions are less effective.

Acknowledgements

We would like to thank the anonymous reviewers for their thoughtful comments and Brave Research team for providing valuable feedback during the project. Panagiotis Tigas is supported by the UK EPSRC CDT in Autonomous Intelligent Machines and Systems (grant reference EP/L015897/1). Zainul Abi Din is supported by a grant from Bouncer Technologies to UC Davis (grant reference A20-2169).

References

[1] Adblock Plus for Chrome support. https://adblockplus.org/.

[2] Annoying online ads do cost business. https://www.nngroup.com/articles/annoying-ads-cost-business/.

[3] Apple neutered ad blockers in Safari . https://www.zdnet.com/article/apple-neutered-ad-blockers-in-safari-but-unlike-chrome-users-didnt-say-a-thing/.

[4] Brave - Secure, Fast & Private Web Browser with Adblocker. https://brave.com/.

[5] Chromium Graphics. https://www.chromium.org/developers/design-documents/chromium-graphics.

[6] Coalition for better ads. https://www.betterads.org/research/.

[7] EasyList. https://easylist.to.

[8] ExpressVPN. https://www.expressvpn.com/.

[9] Facebook Ad placements. https://www.facebook.com/business/help/407108559393196.

[10] Facebook's Arms Race with Adblockers Continues to Escalate. https://motherboard.vice.com/en_us/article/7xydvx/facebooks-arms-race-with-adblockers-continues-to-escalate.

[11] False and deceptive display ads at yahoo's right media, 2009. http://www.benedelman.org/rightmedia-deception/#reg.

[12] Firefox's Enhanced Protection. https://blog.mozilla.org/blog/2019/09/03/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/.

[13] Ghostery – Privacy Ad Blocker. https://www.ghostery.com/.

[14] Google Is Finally Making Chrome Extensions More Secure . https://www.wired.com/story/google-chrome-extensions-security-changes/.

[15] Implement hide-if-contains-snippet. https://issues.adblockplus.org/ticket/7088/.

[16] Introducing native ad blocking feature. . https://blogs.opera.com/desktop/2016/03/native-ad-blocking-feature-opera-for-computers/.

[17] iPhone users can block ads in Safari on iOS 9 . https://www.theverge.com/2015/6/11/8764437/iphone-adblock-safari-ios-9.

[18] More than half of local independent news sites are selling sponsored content. https://www.niemanlab.org/2016/06/more-than-half-of-local-independent-online-news-sites-are-now-selling-sponsored-content-survey/.

[19] Page Fair Ad Block Report. https://pagefair.com/blog/2017/adblockreport/.

[20] Puppeteer: Headless Chrome Node API. https://github.com/GoogleChrome/puppeteer.

[21] Selenium: Web Browser Automation. https://www.seleniumhq.org.

[22] Sentinel: The artificial intelligenence ad detector.

[23] Similar Web. https://www.similarweb.com/.

[24] Skia Graphics Library. https://skia.org/.

[25] The average web page is 3MB. https://speedcurve.com/blog/web-performance-page-bloat/.

[26] uBlock Origin. https://www.ublock.org/.

[27] Why people hate ads? https://www.vieodesign.com/blog/new-data-why-people-hate-ads/.

[28] Karuna Ahuja, Karan Sikka, Anirban Roy, and Ajay Divakaran. Understanding Visual Ads by Aligning Symbols and Objects using Co-Attention. 2018.

[29] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop - AISec '14*, 2014.

[30] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*, 2018.

[31] Claude Castelluccia, Mohamed-Ali Kaafar, and Minh-Dung Tran. Betrayed by your ads! reconstructing user profiles from targeted ads. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies*, PETS'12, page 1–17, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Improving web content blocking with event-loop-turn granularity javascript signatures. *arXiv*, pages arXiv–2005, 2020.

[33] Steven Chen, Nicholas Carlini, and David A. Wagner. Stateful detection of black-box adversarial attacks. *CoRR*, abs/1907.05587, 2019.

[34] Jeremy M Cohen, Elan Rosenfeld, and J Zico Kolter. Certified adversarial robustness via randomized smoothing. In *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 1310–1320. PMLR, 2019.

[35] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016. ACM.

[36] G. Storey, D. Reisman, J. Mayer and A. Narayanan. Perceptual Ad Highlighter. https://chrome.google.com/webstore/detail/perceptual-ad-highlighter/mahgiflleahghaapkboihnbhdplhnchp, 2017.

[37] Kiran Garimella, Orestis Kostakis, and Michael Mathioudakis. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *Proceedings ofWebSci '17*, Troy, NY, USA, 2017. WebSci '17.

[38] Daniel G. Goldstein, R. Preston McAfee, and Siddharth Suri. The cost of annoying ads. In *WWW '13*, 2013.

[39] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*. Springer, 2017.

[40] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Proceedings on Privacy Enhancing Technologies*, volume 2015, 2015.

[41] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017.

[42] Zaeem Hussain, Christopher Thomas, Mingda Zhang, Zuha Agha, Xiaozhong Zhang, Nathan Ong, Keren Ye, and Adriana Kovashka. Automatic understanding of image and video advertisements. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January(14):1100–1110, 2017.

[43] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. 2016.

[44] Umar Iqbal, Zubair Shafiq, Peter Snyder, Shitong Zhu, Zhiyun Qian, and Benjamin Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *CoRR*, abs/1805.09155, 2018.

[45] Harini Kannan, Alexey Kurakin, and Ian Goodfellow. Adversarial Logit Pairing. *arXiv preprint arXiv:1803.06373*, 2018.

[46] J Zico Kolter and Eric Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 1(2), 2017.

[47] Georgios Kontaxis and Monica Chew. Tracking protection in firefox for privacy and performance. *ArXiv*, abs/1506.04104, 2015.

[48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, USA, 2012. Curran Associates Inc.

[49] Alex Kurakin, Dan Boneh, Florian Tramèr, Ian Goodfellow, Nicolas Papernot, and Patrick McDaniel. Ensemble Adversarial Training: Attacks and Defenses. 2018.

[50] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[51] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672. IEEE, 2019.

[52] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, August 2016. USENIX Association.

[53] Bai Li, Changyou Chen, Wenlin Wang, and Lawrence Carin. Certified adversarial robustness with additive noise. In *Advances in Neural Information Processing Systems*, pages 9459–9469, 2019.

[54] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 674–686, New York, NY, USA, 2012. ACM.

[55] Timothy Libert. Exposing the hidden web: An analysis of third-party http requests on 1 million websites. 11 2015.

[56] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[57] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 319–333, 2017.

[58] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples. 2016.

[59] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks against Machine Learning. 2016.

[60] Nicolas Papernot, Patrick Mcdaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, 2016.

[61] Enric Pujol, Tu Berlin, Oliver Hohlfeld, Anja Feldmann, and Tu Berlin. Annoyed users: Ads and ad-block usage in the wild.

[62] Sam Tolomei. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2.

[63] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[64] Ramprasaath R. Selvaraju, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *CoRR*, abs/1610.02391, 2016.

[65] Adblock Sentinel. Adblock Plus, Sentinel, 2018.

[66] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! In *Advances in Neural Information Processing Systems*, pages 3353–3364, 2019.

[67] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. Adversarial training for free! In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3358–3369. Curran Associates, Inc., 2019.

[68] R. Shao, V. Rastogi, Y. Chen, X. Pan, G. Guo, S. Zou, and R. Riley. Understanding in-app ads and detecting hidden attacks through the mobile app-web interface. *IEEE Transactions on Mobile Computing*, 17(11):2675–2688, 2018.

[69] Anastasia Shuba and Athina Markopoulou. NoMoATS: Towards Automatic Detection of Mobile Tracking. *Proceedings on Privacy Enhancing Technologies*, 2020(2), 2020.

[70] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. The Future of Ad Blocking: An Analytical Framework and New Techniques. 2017.

[71] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. Ad-versarial: Defeating perceptual ad-blocking. *CoRR*, abs/1811.03194, 2018.

[72] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019.

[73] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Yan Grunenberger, Konstantina Papagiannaki, Hamed Haddadi, and Jon Crowcroft. Breaking for commercials: Characterizing mobile advertising. In *Proceedings of the 2012 Internet Measurement Conference*, IMC '12, page 343–356, New York, NY, USA, 2012. Association for Computing Machinery.

[74] Antoine Vastel, Peter Snyder, and Benjamin Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *arXiv preprint arXiv:1810.09160*, 2018.

[75] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. A machine learning approach for detecting third-party trackers on the web. In Sokratis Katsikas, Catherine Meadows, Ioannis Askoxylakis, and Sotiris Ioannidis, editors, *Computer Security - 21st European Symposium on Research in Computer Security, ESORICS 2016, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 238–258, Germany, January 2016. Springer Verlag. 21st European Symposium on Research in Computer Security, ESORICS 2016 ; Conference date: 26-09-2016 Through 30-09-2016.

[76] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 2015.

[77] Keren Ye and Adriana Kovashka. ADVISE: Symbolism and external knowledge for decoding advertisements. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11219 LNCS, 2018.

[78] Dinghuai Zhang, Tianyuan Zhang, Yiping Lu, Zhanxing Zhu, and Bin Dong. You only propagate once: Accelerating adversarial training via maximal principle. In *Advances in Neural Information Processing Systems*, pages 227–238, 2019.

[79] Dinghuai Zhang, Tianyuan Zhang, Yiping Lu, Zhanxing Zhu, and Bin Dong. You only propagate once: Accelerating adversarial training via maximal principle. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 227–238. Curran Associates, Inc., 2019.

Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication

Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems (IPADS), Shanghai Jiao Tong University

Abstract

This paper presents UnderBridge, a redesign of traditional microkernel OSes to harmonize the tension between messaging performance and isolation. UnderBridge moves the OS components of a microkernel between user space and kernel space at runtime while enforcing consistent isolation. It retrofits Intel Memory Protection Key for Userspace (PKU) in kernel space to achieve such isolation efficiently and design a fast IPC mechanism across those OS components. Thanks to PKU's extremely low overhead, the inter-process communication (IPC) roundtrip cost in UnderBridge can be as low as 109 cycles. We have designed and implemented a new microkernel called ChCore based on UnderBridge and have also ported UnderBridge to three mainstream microkernels, i.e., seL4, Google Zircon, and Fiasco.OC. Evaluations show that UnderBridge speeds up the IPC by 3.0× compared with the state-of-the-art (e.g., SkyBridge) and improves the performance of IPC-intensive applications by up to 13.1× for the above three microkernels.

1 Introduction

The microkernel OS design has been studied for decades [3, 27, 35, 44, 49, 52, 70]. Microkernels minimize code running in supervisor mode by moving OS components, such as file systems and the network stack, as well as device drivers, into isolated user processes, which achieves good extensibility, security, and fault isolation. Other than the success of microkernels in safety-critical scenarios [1, 40, 66], there is a resurgent interest in designing microkernels for more general-purpose applications, such as Google's next-generation kernel Zircon [3].

However, a cost coming with microkernel is its commonly lower performance compared with its monolithic counterparts, which forces a tradeoff between performance and isolation in many cases. One key factor of such cost is the communication (IPC) overhead between OS components, which is considered as the Achilles' Heel of microkernels [30, 33, 51, 53, 62, 76]. Hence, there has been a long line of research work to improve the IPC performance for microkernels [19, 30, 36, 44, 50, 52, 62, 79, 82]. Through a combination of various optimizations such as in-register parameter passing and scheduling avoidance, the performance of highly optimized IPC has reached less than 1500 cycles per roundtrip [13]. The state-of-the-art SkyBridge IPC design, which retrofits Intel *vmfunc* to optimize IPCs, has further reduced the IPC cost to around 400 cycles per roundtrip [62]. However, such cost is still considerable compared with the cost of invoking kernel components in monolithic kernels (e.g., calling function pointers takes around 24 cycles).

There is always a tension between isolation and performance for OS kernel designs. In this paper, we present a new design named UnderBridge, which redesigns the runtime structure of microkernels to harmonize performance and isolation. The key idea is building isolated domains in supervisor mode while providing efficient cross-domain interactions, and enabling user-space system servers[1] of a microkernel OS to run in those domains. A traditional microkernel OS usually consists of a core kernel in supervisor mode and several system servers in different user processes. With UnderBridge, a system server can also run in an isolated kernel space domain besides a user process. The system servers that run in kernel can interact with each other as well as the core kernel efficiently without traditional expensive IPCs, and applications can invoke them with only two privilege switches, similar to a monolithic OS. Although the number of isolated domains is limited and may be smaller than the number of system servers, UnderBridge supports server migration. The microkernel can dynamically decide to run a server either in a user process or a kernel domain based on how performance-critical it is. However, it is challenging to efficiently provide mutually-isolated domains together with fast cross-domain interactions in kernel space.

Protection Keys for Userspace (PKU [7], Section-2.7, Volume-3), also named as Intel memory protection keys (MPK), has been introduced in recent Intel processors and investigated by researchers to achieve intra-process isolation in user space [29, 39, 64, 77]. As the name "Userspace" indicates, PKU is a mechanism that appears to be only effec-

[1] We name the microkernel OS components implementing system functionalities as system servers. File system and drivers are typical examples.

tive in user space. After a detailed investigation, we observed that *no matter in kernel space (Ring-0) or user space (Ring-3), PKU-capable CPUs transparently enforce permission checks on memory accesses only if the User/Kernel (U/K) bit of the corresponding page table entry is User (means user-accessible)*. Hence, PKU, as a lightweight hardware feature, also offers an opportunity to achieve efficient intra-kernel isolation if all the page table entries for kernel memory are marked with U bit instead of K bit. However, marking kernel memory as user-accessible is dangerous since unprivileged applications may directly access kernel memory. Fortunately, today's OS kernels are usually equipped with kernel-page-table-isolation (KPTI) when preferring stronger security guarantees, including defending against Meltdown-like attacks [20, 55] and protecting kernel-address-space-layout-randomization [8, 42]. User processes and the kernel use different page tables with KPTI, so marking kernel memory as user-accessible in a separate page table does not risk allowing applications to access kernel space.

Hence, UnderBridge allocates an individual page table for the kernel and builds isolated *execution domains* atop MPK memory domains in kernel space. Unlike software fault isolation (SFI), guaranteeing memory isolation with MPK hardware incurs nearly *zero* runtime overhead. Meanwhile, a new instruction, *wrpkru*, can help switching domains by writing a specific register, PKRU (protection key rights register for user pages), which only takes 28 cycles. Thus, domain switches can be quick. With UnderBridge, we design and implement a prototype microkernel named ChCore, which comprises a core kernel and different system servers similar to existing microkernels. ChCore still preserves the IPC interfaces with fast domain switch for the servers but embraces better performance by significantly reducing the IPC costs.

However, we find merely using MPK in kernel fails to achieve the same isolation guarantee as traditional microkernels. On the one hand, since MPK only checks read/write permissions when accessing memory but applies no restrictions on instruction fetching, any server can execute all the code in the same virtual address space. Thus, an IPC gate (a code piece), which is used in UnderBridge to establish the connection between two specific servers, can be abused by any server to issue an illegal IPC. To address the problem, UnderBridge authenticates the caller of an IPC gate through checking its (unique) memory-access permission and ensures the authentication is non-bypassable by validating a secret token at both sides of the IPC gate.

On the other hand, system servers running in kernel space can execute privileged instructions. Although the servers (initially running in user space) should not contain any privileged instruction, such instructions may arise inadvertently on x86, e.g., being composed of the bytes of adjacent instructions. Thus, a compromised server may use return-oriented programming (ROP) [21, 69] to execute them. Traditional microkernels confine system servers in user processes, which, inherently, prevent them from executing privileged instructions and exclude them from the system's trusted computing base (TCB). To do not bloat the TCB, we also prevent the in-kernel system servers from executing any privileged instruction. We leverage hardware virtualization and run ChCore in non-root mode and deploy a tiny secure monitor in root mode. For most privileged instructions that are *not* in the critical path, we configure them to trigger *VMExit* and enforce permission checks in the monitor. For others, we carefully handle them using binary scanning and rewriting.

We have implemented ChCore on a real server with Intel Xeon Gold 6138 CPUs and conducted evaluations to show the efficiency of UnderBridge. To demonstrate the generality of UnderBridge, we have also ported it to three popular microkernels, i.e., seL4 [14], Google Zircon [4], and Fiasco.OC [2]. In the micro-benchmark, UnderBridge achieves 3.0× speedup compared with SkyBridge. In IPC-intensive application benchmarks, UnderBridge also shows better performance than SkyBridge (up to 65%) and improves the performance of the above three microkernels by 2.5×∼13.1×.

In summary, this paper makes the following contributions:

- A new IPC design called UnderBridge that retrofits Intel MPK/PKU in kernel to achieve ultra-low overhead interactions across system servers.
- A microkernel prototype ChCore which uses UnderBridge to move system servers back to kernel space while keeping the same isolation properties as traditional microkernels.
- A detailed evaluation of UnderBridge in ChCore and three widely-used microkernels, which demonstrates the efficiency of the design.

2 Motivation

2.1 Invoking Servers with IPCs is Costly

To obviate the severe consequences incurred by crashes or security breaches [9, 22, 24], the microkernel architecture places most kernel functionalities into different user-space system servers and only keeps crucial functionalities in the privileged kernel, as depicted in Figure-1. Therefore, a fault in a single system server can be caught before it propagates to the whole system.

However, compared with a monolithic OS like Linux, a system service invocation usually becomes more expensive

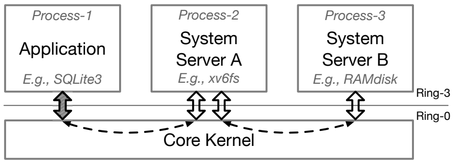
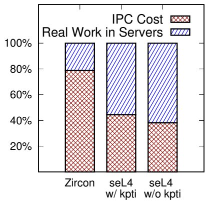


Figure 1: A simplified microkernel architecture. Even without KPTI, calling a server with IPC requires two user-kernel roundtrips with two process switches. A vertical arrow represents one roundtrip, and a dotted line means two process switches.

in such an OS architecture. Figure-1 shows a service invocation procedure that involves two system servers and thus leads to two IPCs. In this case, a microkernel requires *four* roundtrips between user and kernel in total, while Linux only requires *one* (i.e., the leftmost arrow between the application and the OS). It is because Linux invokes different kernel components (like system servers in microkernel) directly through function calls.



(a)

Parts (cycles)	**w/o KPTI**	**w/ KPTI**
Privilege Switch	158	690
Process Switch	295	Included above
Others	277	320
Total	**730** *	**1010**

*The result conforms to the officially reported data (722∼736) [13].

(b)

Figure 2: (a) Invoking servers with IPCs is expensive. (b) A breakdown of seL4 fast-path IPC.

To measure the performance cost of server invocations with IPC, we run SQLite3 [15] on Zircon and seL4 (§ 6 gives the detailed setup). We measure (i) the total time spent on invoking system servers (i.e., an FS server and a RAMdisk server) and (ii) the effective time used in system servers for handling the requests. The difference between the two time durations is considered as the IPC cost. As presented in Figure 2(a), the IPC cost is as high as 79% in Zircon. Even in seL4, which uses highly-optimized IPCs, 44% of the time is spent on IPC when KPTI enabled (38% without KPTI).

Moreover, a system functionality may involve even more IPCs to invoke multiple system servers. For instance, launching an application requires 8 IPC roundtrips (among *Shell, Loader, FS, and Driver*) on Zircon. In contrast, it only needs one or two system calls on Linux (e.g., fork + exec). Therefore, invoking system servers with IPCs is time-consuming in microkernels, which motivates our work in this paper.

2.2 IPC Overhead Analysis

To further understand the overhead of IPC, we break down and measure the cost of each step in the IPC procedure in seL4, which is known to be an efficient implementation of microkernel IPC. Here we use the ideal configuration by referring to [13] and measure a one-way IPC (not a roundtrip) without transferring data.

We find that the direct cost of the IPC consists of three main parts as shown in Figure-2(b). The first part is the privilege switch. A user-space caller starts an IPC by using a *syscall* instruction to trap into the kernel. The kernel needs to save the caller's context, which will be restored when resuming the caller. To invoke the target user-space callee, the kernel transfers the control flow with a *sysret* instruction after restoring the callee's context. The second part is the process switch. The major cost in this part is the *CR3* modification instruction (270 cycles). Since the caller and callee are isolated in different user-space processes (different address spaces), the kernel has to change the address space from the caller to the callee. With KPTI enabled, the kernel further needs to change the address spaces twice during the privilege switch, which inflates the overhead. The third part is other logics in IPC, such as permissions and fast-path conditions checks.

Besides its inherent cost, an IPC will inevitably cause pollution to the CPU internal structures such as the pipeline, instruction, data caches, and translation look-aside buffers (TLB), which has been evaluated in prior work [18, 32, 62, 71]. According to [71], the pollution caused by frequent privilege switches can degrade the performance by up to 65% for SPEC CPU programs.

In summary, the switches of privilege and address space in IPC bring considerable overhead, which motivates our lightweight IPC design to remove these switches.

2.3 Using Intel MPK in Kernel

Background: Intel memory protection keys (MPK) [7] is a new hardware feature to restrict memory accesses. MPK assigns a four-bit domain ID (aka, protection key) to each page in a virtual address space by filling the ID in previously unused bits of page table entries. Thus, MPK can partition user pages within a virtual address space into at most 16 memory domains. To determine the access permissions (read-only, read-write, none) on each memory domain, it introduces a per-core register, *PKRU*, and a new instruction, *wrpkru*, to modify the *PKRU* register in only 28 cycles. It is worth mentioning that MPK checks on memory accesses incur nearly *zero* runtime overhead [39, 77]. Nevertheless, MPK does not enforce permission checks on execution permission. One executable memory page is always executable to any domain, even if *PKRU* forbids the domain from reading the page.

Observation: After a detailed investigation, we observe that no matter in Ring-0 or Ring-3, MPK enforces permission checks on any user-accessible memory page. To enable MPK checks in Ring-0, the *User/Kernel (U/K)* bits of all the corresponding page table entries in a four-level page table have to be set as *User* (i.e., 1). If there exists one entry that contains the *Kernel* bit at any level, MPK will not check the access on the corresponding memory pages. Furthermore, the Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP) should also be disabled for accessing or executing these pages (tagged with *User*) in Ring-0.

2.4 Building Isolated Domains

There are many approaches to build lightweight and isolated domains. SFI (Software Fault Isolation), which has been actively studied over 20 years [34, 46, 60, 68, 80, 87], is one of the most mature candidates. However, although being a general solution to achieve memory/fault isolation, SFI incurs non-negligible runtime overhead due to excessive code instrumentations. For example, two representative studies show that SFI introduces around 15% overhead for SPEC CPU programs on average (Table-2 in [68]), even with the help of the latest boundary-checking hardware MPX (Figure-3 in [46]). Some other approaches [48, 88] uses x86 segmentation for memory isolation, which avoids software checks on memory accesses and is suitable for sandbox execution, but it is not widely used anymore [68].

Instruction	Cost (cycles)
Indirect Call + Return	24
syscall + *sysret*	150
Write *CR3* (no TLB flush)	226
vmfunc (switch EPT)	146
wrpkru	28

Table 1: Cost comparison of selected instructions.

Leveraging advanced hardware features to build isolated domains can achieve better runtime performance. For example, prior work [39, 46, 56, 57, 62, 65, 77] utilizes (extended) page tables to provide isolated domains in user space and exploits instructions like *vmfunc* and *wrpkru* for fast domain switches. We list the costs of these frequently-used instructions in Table-1. The cost of *wrpkru* is the closest to *indirect call*, which is used to invoke kernel components (with function pointers) in monolithic kernels. Therefore, considering that MPK is applicable to the kernel and has good performance property, we propose that MPK can be leveraged to implement an efficient fine-grained isolation mechanism in the kernel.

3 UnderBridge

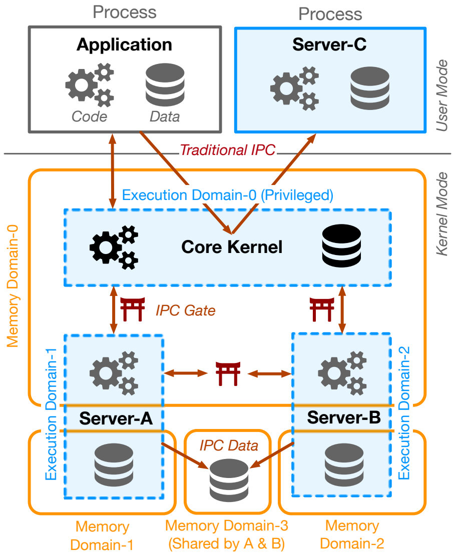


Figure 3: The overview of ChCore based on UnderBridge.

The goal of UnderBridge is to optimize the synchronous IPC[2] while simultaneously maintaining strong isolation. An intuitive design is adopting the MPK-based intra-process isolation [39, 77] to run system servers within an application address space. However, this design has three major problems. First, it requires to map a server into multiple applications' page tables, which makes updating the server's memory mappings especially expensive (i.e., update all the page tables). Second, its cost to setting up the IPC is non-negligible due to intensive page table modifications. Third, it restricts the applications' ability to use the whole address space and the MPK hardware freely.

Instead, UnderBridge boosts IPC performance by putting the system servers, which are user-space processes in traditional microkernels, back into kernel space. Figure-3 shows the system overview. The core kernel resembles the traditional microkernel, which provides crucial functionalities such as managing memory protection, capability enforcement, scheduling, and establishing IPC connections. With UnderBridge, system servers can run in kernel space (e.g., Server-A/B) but are confined in isolated environments (called *execution domain*). UnderBridge makes the core kernel and in-kernel system servers share the same (kernel) address space while leveraging Intel MPK to guarantee memory isolation.

To use MPK in kernel space, UnderBridge tags all the kernel memory pages with "User" bits in the kernel page table, as introduced in § 2.3. However, marking kernel memory as user-accessible enables unprivileged user-space applications to access kernel memory directly. UnderBridge prevents this by allocating a separate page table to each application. An application's page table does not contain the kernel space memory except for a small trampoline region (tagged with "Kernel"), which is used for privilege switch (e.g., *syscall*).

Building IPC connections with in-kernel system servers takes the following steps. First, a system server proactively registers a function address (IPC function) in the core kernel before serving requests. The core kernel will check whether the address is legal, i.e., both executable and belonging to the server. Second, another server can ask the core kernel to establish an IPC connection with the registered server. Third, the core kernel generates an *IPC Gate*, which helps to accomplish the IPC function invocation.

If an application needs to invoke the in-kernel server, it also needs to establish the connection first. Later, it invokes the system call for IPC and traps into the core kernel. Then, the kernel will help to invoke the requested server via the corresponding IPC gate (between the kernel and the server).

3.1 Execution Domains

As shown in Figure-3, UnderBridge constructs isolated *execution domains* over MPK *memory domains* and confines each in-kernel system server in an individual execution domain. Specifically, UnderBridge builds 16 execution domains in kernel space and assigns a unique domain ID (0∼15) to each of them. Execution domain 0 is specialized

[2] Synchronous IPC is commonly used in microkernels, especially when calling system servers. After issuing a synchronous IPC, the caller blocks until the callee returns.

for running the (trustworthy) core kernel and can access all the memory. Every other execution domain (1∼15) has a private memory domain with a specific ID and can only access its private memory domain by default. A system server, exclusively running in one execution domain, stores its data, stack, and heap regions in its private memory domain, which cannot be accessed by other servers. Nevertheless, its code region resides in memory domain 0 that can only be read/written by the core kernel. In this way, the server cannot read-/write its own code but can still execute it (i.e., execute-only memory) as MPK memory domains do not affect instruction fetching. UnderBridge ensures an execution domain can only access allowed memory domains by configuring its *PKRU* register. It also forbids an execution domain (a server) from modifying this register by itself (details in § 4.2).

Shared memory between two servers is allocated by allowing them to access a free memory domain together (e.g., Memory Domain-3). Shared memory between a server and the core kernel is achieved by letting the core kernel directly access the server's private memory domain. Shared memory between an application and a server is achieved by mapping some private memory of the server in the application's page table, which does not require a free memory domain.

3.2 IPC Gates

Even though system servers reside in the same kernel address space, UnderBridge still preserves the well-defined IPC interfaces for them. When connecting two system servers (in two execution domains), the core kernel generates an IPC gate for them, which resides in memory domain 0. Specifically, it first allocates a piece of memory for the gate and loads the gate code, which is small, as shown in Figure-4, into the memory. Then, it fills specific values (e.g., per-gate *SECRET_TOKEN*) into the gate. After that, it gives the gate address to the two system servers connected by this gate.

Later, during an IPC invocation, the gate transfers the control flow from a caller to a callee. To be more specific, it saves the caller's execution state, switches to the callee's domain by setting the *PRKU* register, and restores the callee's execution state. UnderBridge allows the caller and callee to define their calling conventions (the gates only save/restore necessary state by default), which is flexible and efficient. Transferring messages by registers and shared memory are both supported.

Since the system server in UnderBridge only executes when being called through an IPC gate, we adopt the mechanism of decoupling the execution context, which contains the execution state (e.g., register values), with the scheduling context, which contains the scheduling information (e.g., time slice used in the scheduler) [47, 58, 72] and mark those servers as passive. When an application (T1) invokes the system server (T2) through an IPC, T2 inherits T1's scheduling context and then starts executing. When T2 invokes another server (T3), T3 also inherits the scheduling context, which is originally from T1. Thus, the scheduling overhead is avoided in the IPC gates.

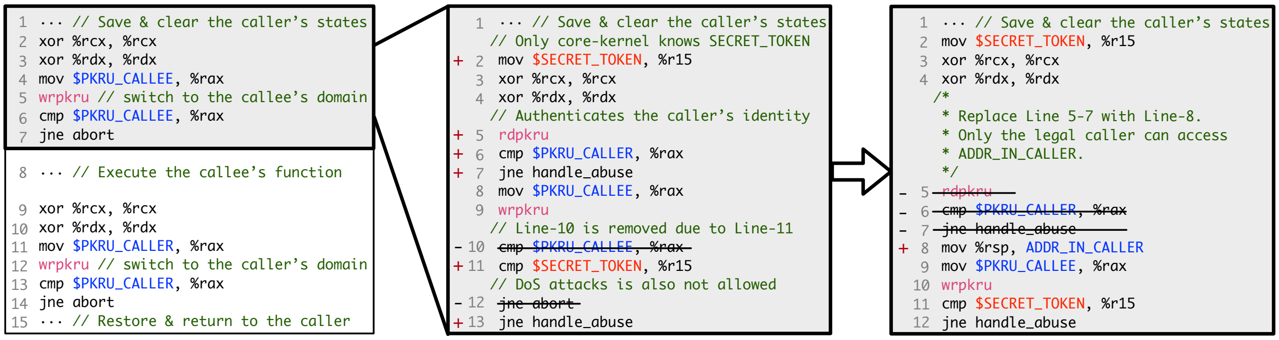
Besides intra-server, IPC gates also exist between the core kernel and the servers, which enables the core kernel to interact with the servers. Specifically, those in-kernel system servers invoke the core kernel's service through dedicated IPC gates connected with the core kernel instead of using *syscall* instructions. To handle exceptions/interrupts during the execution of in-kernel servers, UnderBridge provides similar gates at the beginning of each handler to switch the execution domain to the core kernel (execution domain 0).

3.3 Server Migration

As MPK provides 16 memory domains in total, UnderBridge can only support at most 16 execution domains concurrently, including the reserved one for the core kernel. Nevertheless, more system servers can be required, considering the number of different device drivers. When the number of concurrent system servers exceeds 15, one solution is time-multiplexing [64] but will bring non-negligible overhead if frequently stopping and restarting servers.

Instead, UnderBridge enables *server migration*, which dynamically moves servers between user and kernel space. Each server is compiled as a position-independent executable, and the core kernel assigns disjoint virtual memory regions for different system servers. Whenever runs in an execution domain in the kernel or a user process, a server always uses the same virtual addresses. So, when migrating a server between user and kernel, UnderBridge does not need to do relocations because all the memory references in the server are always valid (no changes), which significantly simplifies the migration procedure. Moreover, the system call layer of the LibC used by servers is also modified, and thus all the *syscall* instructions are organized in one memory page, namely the *syscall page*. When migrating a server from user space to kernel space, the core kernel overrides this page with another prepared page which contains IPC gates connected to the core kernel. Therefore, a server can seamlessly perform system calls no matter in user or kernel space.

Specifically, there are four steps to migrate a server from kernel space to user space. First, the core kernel will wait for the server to enter a quiescent state by blocking new IPC requests to the server temporarily and waiting for the finishes of on-going ones. Second, it will modify the *syscall page* of the server and making the server perform system calls through *syscall* instruction instead of IPC gates later. Third, it will free the execution domain (ID) of the server by setting the domain ID of the server's page table entries to 0 and flush TLBs. If the server installs shared memory with another in-kernel server (S), the domain ID of the shared memory will also be freed because the shared memory can use the domain ID of S in the kernel page table. Last, the core kernel will activate the in-user server and allow clients to issue IPCs to it. Migrating a server from user space to kernel space



```
 1 ··· // Save & clear the caller's states
 2 xor %rcx, %rcx
 3 xor %rdx, %rdx
 4 mov $PKRU_CALLEE, %rax
 5 wrpkru // switch to the callee's domain
 6 cmp $PKRU_CALLEE, %rax
 7 jne abort

 8 ··· // Execute the callee's function

 9 xor %rcx, %rcx
10 xor %rdx, %rdx
11 mov $PKRU_CALLEE, %rax
12 wrpkru // switch to the caller's domain
13 cmp $PKRU_CALLEE, %rax
14 jne abort
15 ··· // Restore & return to the caller
```

(a)

```
   1 ··· // Save & clear the caller's states
    // Only core-kernel knows SECRET_TOKEN
+  2 mov $SECRET_TOKEN, %r15
   3 xor %rcx, %rcx
   4 xor %rdx, %rdx
    // Authenticates the caller's identity
+  5 rdpkru
+  6 cmp $PKRU_CALLER, %rax
+  7 jne handle_abuse
   8 mov $PKRU_CALLEE, %rax
   9 wrpkru
    // Line-10 is removed due to Line-11
- 10 ~~cmp $PKRU_CALLEE, %rax~~
+ 11 cmp $SECRET_TOKEN, %r15
    // DoS attacks is also not allowed
- 12 ~~jne abort~~
+ 13 jne handle_abuse
```

(b)

```
   1 ··· // Save & clear the caller's states
   2 mov $SECRET_TOKEN, %r15
   3 xor %rcx, %rcx
   4 xor %rdx, %rdx
    /*
     * Replace Line 5-7 with Line-8.
     * Only the legal caller can access
     * ADDR_IN_CALLER.
     */
-  5 ~~rdpkru~~
-  6 ~~cmp $PKRU_CALLER, %rax~~
-  7 ~~jne handle_abuse~~
+  8 mov %rsp, ADDR_IN_CALLER
   9 mov $PKRU_CALLEE, %rax
  10 wrpkru
  11 cmp $SECRET_TOKEN, %r15
  12 jne handle_abuse
```

(c)

Figure 4: (a) A basic IPC gate for switching execution domains: Line 1-7 is from the caller to the callee and Line 9-15 is just a reverse process. (b) A security-enhanced IPC gate (from the caller to the callee only) that solves the arbitrary IPC problem. (c) An optimized IPC gate based on the secure one.

works similarly. The mappings of a system server in the kernel page table (used when running in the kernel) will not be removed, and the system server's page table (used when running in user) always exists. And the core kernel will keep corresponding mappings in the two page tables the same.

With server migration, UnderBridge can run frequently-used servers (according to either online or offline profiling) in kernel space while accommodating other servers in user processes. Besides that, we think the number of frequently-used servers may usually be small according to a preliminary survey on some popular applications including Memcached, MySQL, GCC, and a ROS-based (robot) application. We run those applications on Linux and find the most required system calls are only related to the File System, Network, Synchronization, and Memory-Management. For a microkernel, these system calls are usually implemented in only several system servers or directly in the core kernel, which indicates the server migration may rarely happen.

4 Enforcing Isolation in UnderBridge

Threat Model and Assumptions. UnderBridge aims to achieve the same security guarantee as existing microkernels and inherits the same trust model. Specifically, the (trusted) core kernel is assumed to be bug-free and correctly implemented because it has relatively small codebase (e.g., 8,500 LoC in our implementation) and is amenable to formal verification [44]. We do not trust applications or the operating system servers, which may have vulnerabilities or even be maliciously crafted and can be fully compromised by attackers. Physical attacks and hardware bugs are out of the scope of this paper.

Two Security Challenges. Although UnderBridge achieves memory isolation by utilizing MPK hardware, there are still two security threats.

The first threat is the arbitrary IPC problem. The core kernel generates IPC gates in memory domain 0 to ensure any server cannot modify the gates. However, an IPC gate can still be invoked by any (in-kernel) system server as the MPK does not enforce permission check on *execution permission*. Although recent work on MPK-based intra-process isolation [39, 77] does not consider such gate abusing problem, we cannot neglect it because it violates the enforcement of the IPC capability in microkernels. Therefore, UnderBridge ensures only a legal caller (allowed by the core kernel) can successfully use an IPC gate by adding mandatory authentications in the gate. § 4.1 explains the secure design of IPC gates.

The second threat is that untrusted in-kernel system servers run in supervisor mode (Ring-0). Thus, a compromised server can execute any privileged instructions theoretically, which threatens the whole system. For example, it could install a new page table and freely access all memory. One possible defense solution is to enforce control-flow integrity (CFI), which ensures that servers cannot execute any illegal control flow leading to executing privileged instructions. However, CFI instrumentations inevitably bring obvious runtime overhead. Instead, we choose hardware virtualization technology and run ChCore in non-root mode. A tiny secure monitor in root mode audits the execution of most privileged instructions, as summarized in Table 2, by simply trapping them through *VMExits*. In the meanwhile, we use the binary rewriting technique to avoid the expensive *VMExits* on the critical paths. § 4.2 gives more details.

4.1 Unauthorized IPCs Prevention

As shown in Figure-4(a), the responsibility of an IPC gate is saving/restoring the (necessary) execution context of the caller/callee and switching the execution domain from the caller to the callee. Line 2-4 prepares the argument registers for *wrpkru*, which requires *eax* to store the target permission, and both *ecx* and *edx* to be zero. The *wrpkru* instruction in Line 5 sets *PKRU_CALLEE* (the callee's permission) to the *PKRU* register, which specifies the memory-access permission, so the execution domain changes to the callee's domain after this instruction finishes. Line 6-7 prevents a compromised thread from directly jumping (e.g., ROP) to Line 5 with some carefully chosen value in *eax*. Existing work [39, 77] on MPK also designs similar gates for intra-process isolation.

However, such a design faces the gate abusing problem. Since MPK has no restriction on execution permission, a domain can use the gates belonging to other domains to issue IPCs. UnderBridge solves this problem by authenticating the caller's identity in the IPC gates. As the core kernel is responsible for generating IPC gates when two execution domains establish the IPC connection, it knows which domain is the legal caller of the gate. Besides, since each domain has unique memory-access permission, UnderBridge regards the permission as the domain's identity and checks the identity with *rdpkru* in Line 5-7 of Figure-4(b). Moreover, UnderBridge must ensure that the identity check cannot be bypassed. Otherwise, a compromised thread can jump to Line 8/9 without going through the check. To this end, each gate adds two cheap instructions (Line 2 and Line 11 in Figure-4(b)) to guarantee that a successful IPC invocation must go through the check. When setting up an IPC gate, the core kernel randomly produces a 64-bit secret token with *rdrand* instruction and inserts it to the gate. Note that any server cannot read the token value since the IPC gates belong to domain 0 (core kernel). Thus, any caller who wants to pass the check at Line 11 must execute Line 2 first, which ensures the identity check at Line 5-7 is non-bypassable for successful invocations. Similarly, Line 8 is also non-bypassable for successful IPCs; thus, Line 10 is no more required.

Figure-4(c) further gives a more efficient design, which eliminates the overhead of identity check. Although *rdpkru* only takes about 9 cycles, it as well as the extra comparison (Line 6-7) are still on the critical path of IPC. To remove the overhead, UnderBridge authenticates IPC callers by *reusing* the stack-pointer saving instruction (Line 8), which is initially located in the procedure of state saving (Line 1). In this way, any illegal caller will trigger a fault when accessing *ADDR_IN_CALLER* (Line 8) and get caught.

4.2 Privilege Deprivation

In traditional microkernels, system servers only have Ring-3 privilege. To achieve the same security/isolation guarantee, UnderBridge should restrict the servers' behavior when running them in execution domains (Ring-0). However, a compromised server may find and execute privileged instructions at unaligned instruction boundaries with ROP to attack the whole system. Based on an in-depth analysis of privileged instructions (briefly summarized in Table-2), UnderBridge combines virtualization hardware support, binary rewriting technique, and some specific solutions to de-privilege the execution domains for servers with negligible overhead.

According to our survey on four microkernels, system registers such as *IDTR* are only configured at boot time; debug/profile registers are only accessed at debugging/profiling time; most model-specific registers (*MSRs*) are also not accessed in the critical paths. Thus, UnderBridge runs the microkernel in non-root mode and configures the privileged instructions operating on those registers to trigger *VMExits*. And, the secure monitor in root mode checks whether they are executed by the core kernel according to the memory-access permission in *PKRU*. Nevertheless, *rdmsr/wrmsr* may also be used to operate *FS/GS* in the critical path. UnderBridge can configure these two specific registers not to trap or replace them with some newer non-privileged instructions (e.g., *wrfsbase*).

Similarly, since control registers *CR0* and *CR4* are set at boot time only, UnderBridge also traps the setting instructions. Nevertheless, accessing *CR2* and reading *CR0*/*CR4* cannot trigger VMExits. UnderBridge clears *CR2*, where CPU saves the page fault address in the fault handler to prevent information leakage, and hides the real *CR0*/*CR4* value with a shadow value by leveraging virtualization hardware functionality. As for *CR3*, it points to page tables and needs modifications when switching address spaces, which frequently appears in the critical path. So, triggering *VMExits* on *CR3* modifications is expensive. Instead, we use the following lightweight solution (*a special method*). When loading system servers, UnderBridge leverages binary scanning and rewriting to guarantee the servers contain no *CR3* modification instructions, including at unaligned instruction boundaries. While in the core kernel, this privileged instruction must exist to switch address spaces. UnderBridge prevents a compromised system server from executing this instruction in the core kernel through defenses in depth. First (a simple defense), the instruction location is unknown to the servers. Second, to achieve higher security, the core kernel can write this instruction right before executing it and immediately remove it after the execution. Thus, a system server cannot execute it even if knowing its location. Furthermore, on different cores, UnderBridge makes the page table mapping for this instruction (page) different. So, when one core writes this instruction, other cores still cannot execute it.

Instructions that invalidate cache/TLB may be used by compromised servers for conducting performance attacks. While trapping cache invalidation instructions via *VMExits* does not affect overall performance since they are rarely executed, flushing TLB frequently appears on the critical path. So, we use binary rewriting to ensure there are no TLB flush instructions in system servers instead of trapping these instructions. The core kernel must contain these instructions, but they cannot be abused by faulting servers because we make sure they are followed by instructions that access the core kernel memory. Cacheline flush instructions (non-privileged instructions, e.g., *clflush* and *clflushopt*) are also considered because the system servers share the same address space with the core kernel. Nevertheless, these instructions obey MPK memory (read) checks and thus, cannot be utilized by a server to flush others' memory areas.

For other privileged instructions and I/O related operations, we take similar solutions as listed in Table-2. One thing to notice is that a compromised server may disable interrupts

Categories	Related Instructions or Registers	Usages in Zircon/Fiasco.OC/seL4/ChCore or Brief Explanations	Solutions
Load/Store System Registers	IDTR, GDTR, LDTR TR, XCR0 ...	Although seL4 uses "LTR" instruction when switching processes, it can be removed by setting different TSSs at boot time as do in other microkernels. Others are required at boot time only.	VMExit
Debug/Profile Registers	Debug registers RDPMC	Required for debugging and profiling, which are not performance-oriented.	VMExit
Model Specific Registers	RDMSR/WRMSR	Usually, they are mostly used at boot time or debug time and can trigger VMExits for selected registers according to configuration bitmaps.	Selected VMExit
Read/Write Control Registers	mov CRn, reg mov reg, CRn CLTS (modify CR0)	- In the four microkernels, CR0/CR4 is written at boot time only and CR8 is not used. - No VMExits: accessing CR2, reading CR0 and CR4. - CR3 is used for switching address space. So, we handle it with a special method.	- VMExit - Clear/hide - Special method
Cache/TLB States	- INVD/WBINVD - INVLPG/INVPCID - clflush instructions	- Whole cache eviction. WBINVD is rarely used, and INVD is even not used. - TLB clear is needed after updating page tables. So, triggering VMExits is costly here. - (executable in Ring-3) Evicting a single cache line. MPK checks take effects.	- VMExit - Binary rewritng - None
I/O Related Operations	- Port I/O - MMIO - DMA	- Port I/O is not performance-oriented and can trigger VMExits with I/O bitmaps. - MMIO operations go through MPK checks. - The core kernel initializes DMA devices at boot time and takes the control plane.	- VMExit - None - None
Other Privilege Instructions	- SMSW, RSM, HLT ... - SWAPGS, SYSRET... - CLI, POPFQ...	- Either related to other modes like legacy and SMM mode or rarely used. - Cannot break the system states, otherwise leading to the execution of fault handlers. - Can be used by compromised servers to disable interrupts.	- VMExit - None - Check in VMM
PKRU Register (Ring-0/3)	- xsave set instructions - WRPKRU	- For restoring extended processor states, which may include PKRU state. - For changing the value of PRKU register as used in IPC gates.	- No restoring - Binary rewriting

Table 2: Deprive the execution domains for system servers of the ability to execute privileged instructions.

through instructions like *cli* to monopolize the CPU. Fortunately, it cannot disable the host timer interrupts, which unconditionally trigger *VMExits*. Thus, the secure monitor can easily detect such malicious behaviors by checking *PKRU* and *interrupt state*.

Last but not least, we must forbid system servers from changing the *PKRU* register, i.e., changing the memory-access permission, by themselves. There are two kinds of instructions that can modify *PKRU*. For the first kind (*xrstor/xrstors*), the core kernel configures them not to manage *PKRU* by setting a control bit (bit-9) in *XRC0*. For the second kind (*wrpkru*), the core kernel ensures it does not exist outside the IPC gates by rewriting the binary code (similar to [77]). The *wrpkru* in the IPC gates cannot be abused as specified in § 4.1.

We omit the detailed policies of the binary rewriting in this paper as it is a mature technique [17, 25, 77]. Nevertheless, it is worth noting that using binary rewriting to directly eliminating all privilege instructions is undecidable because some privilege instructions only take one byte (e.g., hlt). Our hybrid approach is both effective and efficient.

4.3 Security Analysis

By introducing in-kernel servers, our system has one major difference from existing microkernels, which may lead to a larger attack surface. The in-kernel servers run in the kernel mode, which means a compromised server is able to execute any privileged instruction. We will analyze the attacks caused by the difference and illustrate how to defend against them.

Restricting privileged in-kernel servers: System servers are not trusted in our threat model. Although they can run in the kernel mode, they are highly restricted when trying to attack the core kernel, other servers, or the applications.

We assume that an attacker has fully compromised an in-kernel server and can execute arbitrary instructions. Since the server runs in another execution domain (no access to the memory domain 0), it cannot directly access the memory of the core kernel. As long as it tries to read or write any disallowed memory, a CPU exception will immediately be triggered and handled by the core kernel.

There are four ways to bypass the memory isolation mechanism enforced by MPK: the first one is to run the disabling instructions, e.g., by setting *CR4.PKE* to 0 or setting *CR0.WP* to 0; the second is to change the *PKRU* register to gain access permission of other memory domains illegally; the third is to change the page table base address by setting the *CR3* register; the fourth is to modify the page table directly.

ChCore can defend against all these attacks. Before loading a server, ChCore uses binary scanning/rewriting to eliminate the undesired privileged instructions. At runtime, the secure monitor will prevent a server from executing other privileged instructions. Compared with running on traditional microkernels, servers, including maliciously crafted ones, have no more attack means on microkernels with UnderBridge. First, when the malicious server executes the disabling instructions, it will trigger *VMExits*, and the monitor will locate the compromised server. Second, as described in the last paragraph of § 4.2, the two ways of modifying the *PKRU* register are prevented. Third, the malicious server has no way to modify *CR3* since the binary rewriting guarantees no *CR3* modification instructions exist in any server's address space. Fourth, the malicious server cannot modify the page table because the kernel page table resides in memory domain 0. Meanwhile, it cannot modify or add instructions which require to change the page table first. The isolation between in-kernel servers is the same as the isolation between an in-kernel server and the core kernel. Since an in-kernel server does not share address space with user applications, it cannot access applications' memory either.

Defending side-channel attacks: Since all the in-kernel servers share one address space, it is easier for a malicious one to issue Spectre [45] and Meltdown [54] attacks compared with the case where all servers have their own address spaces. Although these attacks are caused by CPU bugs (out-of-scope), ChCore can mitigate them with existing software defenses like using address randomization makes a compromised server hard to locate the sensitive memory area. Considering the secret tokens leakage on buggy CPUs, extra checks can be added in the IPC gates, e.g., Line-10 in Figure-4(b), to prevent malicious PKRU modification and ensure the memory isolation. Besides that, most known hardware vulnerabilities have been fixed by major CPU vendors in their latest products [5, 6], which is orthogonal to ChCore.

5 Implementation

Based on our UnderBridge design, we have implemented a prototype microkernel ChCore, which contains about 8500 lines of C code (LOC). ChCore runs in guest-mode, i.e., non-root mode on x86_64, and a small secure monitor (around 300 LOC) runs in hypervisor-mode, i.e., root mode on x86_64. We have implemented the tiny secure monitor in a minimal virtualization-layer, RootKernel (1,500 LOC) of SkyBridge [62]. Note that RootKernel is not a hypervisor and works only for running one OS in the guest-mode and thus avoids most overhead caused by virtualization. Therefore, although our system requires hardware virtualization, it still can be deployed in bare-metal machines with RootKernel and achieve close-to-native performance. Considering the above, our system increases the trusted computing base (TCB) by 1,800 LOC in total when running on bare-metal machines, which is acceptable. Besides, we also integrate the tiny secure monitor in a commercial hypervisor, KVM, which makes deploying our system in cloud feasible. Even if nested virtualization is required, our secure monitor can still work because it simply utilizes the hardware-provided capability to trap sensitive instructions. Because the instructions to trap are deliberately selected and do not exist on critical paths, they will not degrade the overall performance. Nevertheless, our current implementation requires cloud providers to patch their hypervisors. In such a case, our system increases TCB by 300 LOC. Alternatively, we may leverage eBPF [61, 81] to deploy our secure monitor without modifications to the commercial hypervisor (left as future work).

We also apply UnderBridge to three state-of-the-art microkernels, i.e., seL4 [3], Zircon, and Fiasco.OC, to demonstrate the generality of the design. The porting effort is about 1000∼1500 LOC for each of them. Since UnderBridge uses different page tables for applications and kernel, it also enables and leverages the PCID hardware feature for avoiding unnecessary TLB flushing. As native Fiasco.OC does not support to use this feature, we also add a simple extension to assign different PCIDs to an application and the kernel.

6 Performance Evaluation

Basic Setup. We conduct all the experiments on a Dell PowerEdge R640 server, which is equipped with a 20-core Intel Xeon Gold 6138 2.0GHz CPU and 128GB memory. Both Turbo Boost and Hyper-threading are disabled. ChCore runs on Linux/KVM-4.19 and QEMU-4.1.

Systems for Comparison. We evaluate the native IPC performance of three popular microkernels (Zircon, seL4, and Fiasco.OC) on bare metal. Also, we evaluate SkyBridge [62], which is the state-of-the-art optimization for IPC in microkernels by using *vmfunc*. SkyBridge deploys a small hypervisor called RootKernel and runs microkernels in non-root mode. When evaluating UnderBridge, we deploy the secure monitor of UnderBridge in RootKernel and run microkernels with UnderBridge on it. As Zircon has no kernel-page-table-isolation (KPTI) support, we simulate the overhead of page table switching by writing *CR3* twice when evaluating UnderBridge on it, this is because UnderBridge requires to make the kernel and applications use different page tables.

6.1 IPC Performance Analysis

Cross-server IPC. Firstly, we analyze the IPC performance between two servers (we abbreviate "system server" as "server" in this section) in a micro-benchmark, which uses *rdtsc* instruction to measure the *round-trip* latency of invoking an empty function in server B from server A.

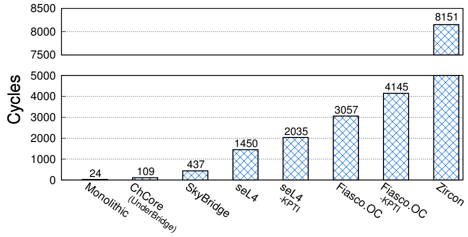


Figure 5: Round-trip latency of cross-server IPC.

Figure-5 gives the absolute cost of cross-server IPCs in different designs. Since invocations between servers (components) in monolithic kernels are usually achieved by using (indirect) *call*/*ret* instructions, the round-trip latency is only 24 cycles. The latency of an IPC round-trip in ChCore is 109 cycles, which is dominated by two *wrpkru* instructions (56 cycles in total). Note that the IPC is achieved by UnderBridge and thus only involves the lightweight *wrpkru* and the procedure of saving necessary registers. SkyBridge requires two much heavier *vmfunc* instruction (292 cycles in total) and therefore has larger latency.

The round-trip latency of a native IPC in the other three microkernels are much more noticeable. Among them, seL4

[3] We do not retain formal correctness guarantees of seL4.

shows the best performance as it will directly switch the caller thread to callee thread when executing fast-path IPCs. Although Fiasco.OC applies a similar strategy, it has a more complex IPC capability handling procedure. Zircon does not support direct-switch and thus has the worst performance due to the high scheduling cost.

Compared with the native IPC in the three microkernels, our UnderBridge design is more than 12.3× faster. The performance improvement mainly comes from two parts. First, UnderBridge avoids the time-consuming privilege switches in traditional IPC designs, as measured in § 2.2. Second, UnderBridge avoids the complex validation and invocation of the IPC capability on the critical path. An IPC gate is only generated after the corresponding capabilities having been checked, so UnderBridge needs not to check the capability at runtime. Also, the gate only requires several lightweight instructions for the domain switching (details in § 4.1).

Another benefit of UnderBridge is that the in-kernel servers can invoke system calls faster via the IPC gates instead of using *syscall* instructions.

Application-to-Server IPC. We further analyze the IPC (round-trip) performance between a user application and a server in this part. Commonly, a system call involves multiple servers in microkernels (or multiple kernel components in monolithic kernels), which means an application-to-server IPC may involve several cross-server IPCs. To simulate such cases, we design a micro-benchmark that includes one application and several servers. Each server will do nothing but routing IPCs to another server. We vary the number of cross-server IPCs in the benchmark.

Approaches	Cross-server IPCs		
	0	1	2
SkyBridge	437	931	1390
ChCore (UnderBridge)	723	856	981
seL4	1450	2932	4266

Table 3: Round-trip latency (cycles) of one application-to-server IPC and different number of cross-server IPCs.

Table-3 compares the performance of SkyBridge (applied on Fiasco.OC) and UnderBridge (applied on ChCore) in this micro-benchmark. Results are similar when we apply them to other microkernels and thus omitted. If an application invokes a server *without* causing any cross-server IPCs, SkyBridge (437 cycles) shows better performance than UnderBridge (723 cycles). It is because UnderBridge has to switch the privilege level and the address space for transferring the control flow from user space to kernel space, while SkyBridge applies more lightweight EPT switching via *vmfunc*. Nevertheless, UnderBridge still outperforms the best native IPC (1450 cycles in seL4), because of reducing one privilege switch and minimizing the software logic of an IPC.

As the number of cross-server IPCs increases, the latency of SkyBridge increases in proportion. It is because that IPC from application to the server and between servers are symmetric and cost the same cycles. In contrast, the latency of UnderBridge grows much slower because its cross-server IPC is much more lightweight. As shown in Table-3, the performance of UnderBridge becomes better than SkyBridge when involving one cross-server IPC, and is better than that by 42% when involving two. The performance speedup is expected to grow along with the increasing number of the cross-server IPCs and finally close to 3.0× as in Figure-5.

6.2 Application Benchmarks

We further evaluate the performance of UnderBridge with two real-world applications: a database application and an HTTP server application. In the following experiments, shared memory is used to transfer data during IPCs.

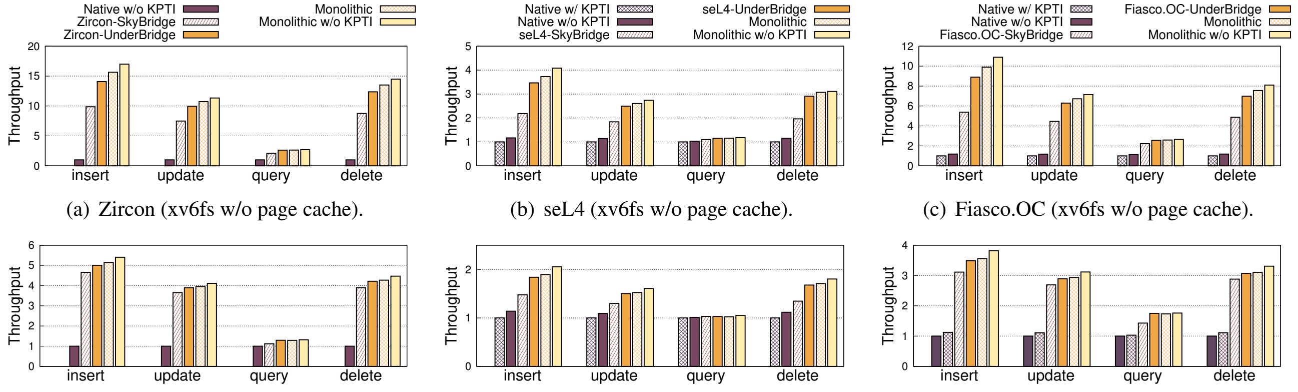
Database Evaluation. To faithfully compare with SkyBridge, we use the benchmarks in [62]. Specifically, for serving a relational database, SQLite3 [15], we run two system servers: one is a file system named xv6fs [16, 23], and the other is a RAMdisk (memory-only). When SQLite3 operates on a file, it will first invoke the xv6fs server by an application-to-server IPC, and then the xv6fs will read or write the RAMdisk by cross-server IPCs.

Basic Operations. We first evaluate the performance of basic operations, including insert, update, query, and delete operations. Our evaluation includes three IPC approaches: the native IPC, UnderBridge, and SkyBridge. Specifically, for each microkernel, we not only evaluate the performance with its native IPC designs but also test the performance after applying UnderBridge and SkyBridge to it. We also emulate the performance of a monolithic kernel by replacing all the IPC gates in UnderBridge with function calls.

Figure-6(a), 6(b) and 6(c) show the normalized throughput of basic operations in the three microkernels, separately. The baseline is set as the performance of native IPC design in each microkernel.

UnderBridge achieves up to 13.1×, 9.0×, 1.6× and 11.3× speedup for each of these operations, individually. The improvement of query operations is relatively small since SQLite3 has an internal buffer for storing recent data and may handle the queries without issuing IPCs. Compared with SkyBridge, the performance improvement of UnderBridge (up to 65%) is because a single IPC from SQLite3 to xv6fs is likely to trigger multiple cross-server IPCs between xv6fs and RAMdisk. Even compared with the emulated performance of monolithic kernels, UnderBridge only introduces about 5.0% overhead.

The above-tested xv6fs (exactly the same one used in the SkyBridge paper [62]) contains no page cache, which, thus, emulates an IPC-intensive scenario. We further enable the page cache in xv6fs to show how UnderBridge performs with fewer cross-server IPCs between xv6fs and RAMdisk. As shown in Figure-6(d), 6(e) and 6(f), the performance improvement of UnderBridge is still obvious. UnderBridge shows up to 4.0×, 2.9×, 0.7× and 3.2× speedup



(a) Zircon (xv6fs w/o page cache).

(b) seL4 (xv6fs w/o page cache).

(c) Fiasco.OC (xv6fs w/o page cache).

(d) Zircon (xv6fs w/ page cache).

(e) seL4 (xv6fs w/ page cache).

(f) Fiasco.OC (xv6fs w/ page cache).

Figure 6: Normalized throughput of basic SQLite3 operations.

compared with the native IPC and achieves comparable performance with monolithic kernels. Nevertheless, since fewer cross-server IPCs are involved, the maximum improvement of UnderBridge compared with SkyBridge drops from 65% to 25%.

Since UnderBridge runs in non-root mode, we also count the number of *VMExit*, which is known as the cost of virtualization. Thanks to the careful design of RootKernel and our secure monitor, there are *almost zero VMExits* during the experiments. For example, at most one VMExit (due to timer) happens during the query test.

YCSB Benchmark. We also evaluate SQLite3 against YCSB workloads. Figure-7(a) and 7(b) show the normalized throughput of YCSB-A (50% update and 50% query) with the page cache disabled and enabled in xv6fs, separately. We use the same baseline as the basic operation evaluation. Even for seL4, which is the most efficient among the three microkernels, UnderBridge improves the application's throughput from 35% to 105%. UnderBridge also brings a better performance (from 7% to 35%) than SkyBridge. Besides, it is only slightly slower (3.3% on average) than the monolithic kernel. Other YCSB workloads give similar results.

Furthermore, Figure-7(c) gives a detailed analysis of the experiments with page cache *enabled* in xv6fs. First, the ratio of IPCs from SQLite3 to xv6fs (application-to-server) and xv6fs to RAMdisk (cross-server) is about 1:2. Thus, UnderBridge outperforms SkyBridge, according to Table-3. Second, it helps to reduce the ratio of time spent on IPCs to around 11% while the other three microkernels spend at least 30% of the time on IPCs. Third, it makes the application and the servers execute faster (about 10%) owing to less indirect costs such as cache/TLB pollution.

Server Migration. We also evaluate the performance of server migration, although it should rarely happen. We still run SQLite3, xv6fs, and RAMdisk as above on Zircon and trigger the server migration. Taking RAMdisk (128 MB virtual memory range) as an example, migrating it from kernel to user takes about 84,361 cycles. Most of the cycles (83,517) are spent on modifying the kernel page table to free the domain ID (i.e., the third step for migrating a server specified in § 3.3). Migrating RAMdisk from user to kernel takes more cycles (90,189) mainly because more cycles are spent on waiting for finishes of on-going IPCs.

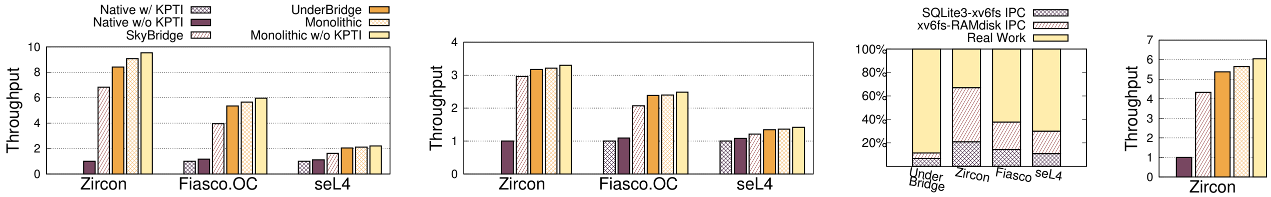
HTTP Server Evaluation. For running an HTTP server (a user-space application), we create three system servers: a socket server, a TCP/IP protocol stack server, and a loopback network device driver (not involving the real network device) atop lwIP [31] library. We measure the throughput of a simple HTTP server from the client-side, which receives requests from the network and sends back a static HTML page.

We perform this evaluation on Zircon. As shown in Figure-7(d), UnderBridge improves the throughput of the HTTP server by 4.4×. We also implement the same benchmark with SkyBridge. UnderBridge outperforms SkyBridge by about 24% because a network request also triggers multiple cross-server IPCs.

7 Related Work

Reconstructing monolithic kernels. The development of monolithic kernels follows the philosophy of modularization, but all the kernel components are not isolated from each other. With reliability and security attracting a fair amount of attention, we witness interest in reconstructing monolithic kernels to achieve better fault isolation and higher security [17, 25, 26, 38, 41, 59, 63, 65, 67, 74, 75, 84, 86]. Dautenhahn et al. [25] build one memory protection domain inside the BSD kernel and run a small trusted kernel in that domain to control memory mappings. Mondrix [84] implements a compartmentalized Linux kernel with eleven isolated modules based on customized security hardware [83]. Proskurin et al. [65] propose to use Intel EPT and *vmfunc* to isolate critical kernel data in different domains. Nooks [74] and LXFI [59] focus on improving the reliability of Linux by isolating kernel modules, especially device drivers.

Our work does share some similarities with prior work on intra-(monolithic)kernel isolation. Nevertheless, we fo-



(a) YCSB-A (xv6fs w/o page cache).
(b) YCSB-A (xv6fs w/ page cache).
(c) Breakdown analysis.
(d) HTTP server.

Figure 7: (a), (b), and (d) share the same legend. (a) and (b) show the normalized throughput of YCSB-A with xv6fs's page cache disabled and enabled separately. (c) shows the time breakdown of YCSB-A benchmark. (d) demonstrates the normalized throughput of a HTTP server.

cus on accelerating IPCs for microkernel architectures while maintaining strong isolation (both ends). We need to do little modification/instrumentation on system servers of a microkernel. This is because system servers of a microkernel are designed to run in different user processes, and all the interactions are explicit IPCs, which is different from the subsystems in Linux (no clear boundaries and have complex shared memory references). We achieve intra-kernel isolation by retrofitting Intel MPK, which is lightweight and commercially-available, to build multiple execution domains in kernel space. Furthermore, UnderBridge may also be generalized to other kernel scopes with more efforts in the future. On one side, the proposed abstraction of the execution domain can be extended to accommodate different kernel modules in monolithic kernels, and the IPC gates can still be used to handle interactions between those modules. On the other side, our design can also be applied in kernels written in memory-safe languages [11, 24] to isolate some unsafe code (e.g., the code with "unsafe" tag in Rust).

Accelerating IPCs. Optimizing the performance of IPC in microkernels is continuously studied for a long time [19, 36, 44, 50, 52, 82]. For example, LRPC [19] eliminates the scheduling overhead during an IPC by using the thread-migration model [36, 37]; seL4 [44] provides the fast-path IPC, which also avoids scheduling and passes arguments through registers. Nevertheless, even with these software-based optimizations, IPC-intensive applications on microkernels still suffer from the IPC overhead.

Recent studies present new designs to accelerate IPCs with advanced hardware features. SkyBridge [62] utilizes the *vmfunc* to allow a process to invoke a function in another process directly without the kernel's involvement. XPC [30] is a hardware proposal that accelerates IPCs by implementing efficient context switch and memory granting. dIPC relies on another hardware proposal [78] to put processes into the same address space and thus make IPCs faster. Our design shows better performance than SkyBridge and requires no hardware modification.

Usage of Intel MPK. Intel MPK/PKU has been utilized by [39, 64, 77] to achieve efficient intra-process isolation, which can build mutual-distrusted execution environments in a single user process. There are two main differences between UnderBridge and them. First, UnderBridge retrofits MPK that designed for user space in kernel space to improve the IPC performance for microkernels (the first effort to our knowledge) and also faces more security challenges. Applications can still utilize MPK in user space as they want since they have different page tables from the kernel. Second, UnderBridge authenticates the caller of each MPK gate for enforcing IPC capability to prevent illegal domain switches, while prior work allows arbitrary domain switches.

There exist two concurrent studies [38, 73] to UnderBridge. They also propose to utilize MPK in kernel mode but with different goals and designs. IskiOS [38] leverages MPK to defend against code-reuse attacks (e.g., protecting shadow stacks) in the kernel. Sung et al. [73] uses MPK to enhance the isolation for a unikernel while does not solve the security challenges identified in UnderBridge.

MPK-like features on other architectures. Tagged memory [28, 43, 85], which can provide MPK-like features, is added in other architectures, which brings the potential to make UnderBridge more general to those architectures. Recently, the RISC-V security community also considers enhancing the PMP (physical memory protection) isolation with the tagged memory mechanism [12]. However, UnderBridge cannot work on current ARMv8 (aarch64). Yet, ARM v8.5 has included memory tagging extensions [10], by extending which with more mechanisms, we may provide a similar mechanism workable on future ARM platforms.

8 Conclusion

This paper introduces UnderBridge, a redesign of the runtime structure of microkernel OSes for faster OS services. To demonstrate UnderBridge's efficiency, we have built a prototype microkernel named ChCore and ported it to three existing microkernels. Performance evaluations showed that UnderBridge can achieve better performance in IPC-intensive workloads compared with prior work.

9 Acknowledgement

We sincerely thank our shepherd Antonio Barbalace and the anonymous reviewers for their insightful suggestions. This work is supported in part by China National Natural Science Foundation (No. 61925206, 61972244, and U19A2060) and a grant from the Science and Technology Commission of Shanghai Municipality (No. 19511121100). Yubin Xia is the corresponding author.

References

[1] Apple ios security-ios 12.1. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf. Referenced December 2019.

[2] Fiasco.oc repository. https://l4re.org/download/snapshots/. Referenced December 2019.

[3] Fuchsia. https://fuchsia.dev/fuchsia-src. Referenced December 2019.

[4] Fuchsia repository. https://fuchsia.dev/fuchsia-src/development/source_code. Referenced December 2019.

[5] Ian cutress: Analyzing core i9-9900k performance with spectre and meltdown hardware mitigations. https://www.anandtech.com/show/13659/analyzing-core-i9-9900k-performance-with-spectre-and-meltdown-hardware-mitigations. Referenced December 2019.

[6] Intel corporation. engineering new protections into hardware. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html. Referenced December 2019.

[7] Intel software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. Referenced December 2019.

[8] Kernel page table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation. Referenced December 2019.

[9] Linux kernel cves. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Referenced December 2019.

[10] Memory tagging in armv8.5-a. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a. Referenced May 2020.

[11] Redox operating system. https://www.redox-os.org/. Referenced December 2019.

[12] Risc-v isa specification. https://riscv.org/specifications/. Referenced May 2020.

[13] sel4 performance report. http://sel4.systems/About/Performance/. Referenced December 2019.

[14] sel4 repository. https://github.com/seL4/seL4. Referenced December 2019.

[15] Sqlite. https://www.sqlite.org/index.html. Referenced December 2019.

[16] xv6 repository. https://github.com/mit-pdos/fscq/tree/master/xv6. Referenced December 2019.

[17] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[18] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[19] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[20] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 769–784, New York, NY, USA, 2019. ACM.

[21] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[22] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[23] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[24] Cody Cutler, M Frans Kaashoek, and Robert T Morris. The benefits and costs of writing a {POSIX} kernel in a high-level language. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 89–105, 2018.

[25] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth In-*

ternational Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[26] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *NDSS*, 2017.

[27] Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. Curios: improving reliability through operating system structure. pages 59–72, 2008.

[28] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2014.

[29] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. ACM.

[30] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. Xpc: Architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 671–684, New York, NY, USA, 2019. ACM.

[31] Adam Dunkels. lwip-a lightweight tcp/ip stack. *Available from World Wide Web: http://www. sics. se/ adam/lwip/index. html*, 2002.

[32] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[34] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[35] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, New York, NY, USA, 1996. ACM.

[36] Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 9–9, Berkeley, CA, USA, 1994. USENIX Association.

[37] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association.

[38] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Iskios: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[39] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 489–503, Berkeley, CA, USA, 2019. USENIX Association.

[40] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[41] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. Lightweight capability domains: Towards decomposing the linux kernel. *SIGOPS Oper. Syst. Rev.*, 49(2):44–50, January 2016.

[42] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[43] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[44] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[45] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin,

Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[46] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452, New York, NY, USA, 2017. ACM.

[47] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[48] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1441–1454, New York, NY, USA, 2018. Association for Computing Machinery.

[49] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[50] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[51] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[52] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[53] Jochen Liedtke. A persistent system in real use - experiences of the first 13 years. pages 2 – 11, 01 1994.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 973–990, Berkeley, CA, USA, 2018. USENIX Association.

[56] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 49–64, Berkeley, CA, USA, 2016. USENIX Association.

[57] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619, New York, NY, USA, 2015. ACM.

[58] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 26:1–26:16, New York, NY, USA, 2018. ACM.

[59] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[60] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[61] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[62] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.

[63] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[64] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 241–254, Berkeley, CA, USA, 2019. USENIX Association.

[65] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Poly-

chronakis. xmp: Selective memory protection for kernel and user space. In *Proceedings of 41st IEEE Symposium on Security and Privacy*, S&P '20, 2020.

[66] Franklin Reynolds. An architectural overview of alpha: A real-time, distributed kernel. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Berkeley, CA, USA, 1992. USENIX Association.

[67] O. Schwahn, S. Winter, N. Coppik, and N. Suri. How to fillet a penguin: Runtime data driven partitioning of linux code. *IEEE Transactions on Dependable and Secure Computing*, 15(6):945–958, Nov 2018.

[68] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[69] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[70] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[71] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.

[72] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[73] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 143–156, 2020.

[74] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[75] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. A policy-centric approach to protecting os kernel from vulnerable lkms. *Software: Practice and Experience*, 48(6):1269–1284, 2018.

[76] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Experimental Computer Science on Experimental Computer Science*, ecs'07, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[77] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1221–1238, Berkeley, CA, USA, 2019. USENIX Association.

[78] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. Codoms: Protecting software with code-centric memory domains. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480. IEEE, 2014.

[79] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 16–31, New York, NY, USA, 2017. ACM.

[80] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[81] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 33–47, 2014.

[82] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5):38–49, September 2016.

[83] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[84] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*,

SOSP '05, pages 31–44, New York, NY, USA, 2005. ACM.

[85] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[86] Xi Xiong, Donghai Tian, Peng Liu, et al. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, volume 11, 2011.

[87] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[88] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Simon Shillaker
Imperial College London

Peter Pietzuch
Imperial College London

Abstract

Serverless computing is an excellent fit for big data processing because it can scale quickly and cheaply to thousands of parallel functions. Existing serverless platforms isolate functions in ephemeral, stateless containers, preventing them from directly sharing memory. This forces users to duplicate and serialise data repeatedly, adding unnecessary performance and resource costs. We believe that a new lightweight isolation approach is needed, which supports sharing memory directly between functions and reduces resource overheads.

We introduce *Faaslets*, a new isolation abstraction for high-performance serverless computing. Faaslets isolate the memory of executed functions using *software-fault isolation* (SFI), as provided by *WebAssembly*, while allowing memory regions to be shared between functions in the same address space. Faaslets can thus avoid expensive data movement when functions are co-located on the same machine. Our runtime for Faaslets, FAASM, isolates other resources, e.g. CPU and network, using standard Linux *cgroups*, and provides a low-level POSIX host interface for networking, file system access and dynamic loading. To reduce initialisation times, FAASM restores Faaslets from already-initialised snapshots. We compare FAASM to a standard container-based platform and show that, when training a machine learning model, it achieves a 2× speed-up with 10× less memory; for serving machine learning inference, FAASM doubles the throughput and reduces tail latency by 90%.

1 Introduction

Serverless computing is becoming a popular way to deploy data-intensive applications. A function-as-a-service (FaaS) model decomposes computation into many functions, which can effectively exploit the massive parallelism of clouds. Prior work has shown how serverless can support map/reduce-style jobs [42, 69], machine learning training [17, 18] and inference [40], and linear algebra computation [73, 88]. As a result, an increasing number of applications, implemented in diverse programming languages, are being migrated to serverless platforms.

Existing platforms such as Google Cloud Functions [32], IBM Cloud Functions [39], Azure Functions [50] and AWS Lambda [5] isolate functions in ephemeral, stateless *containers*. The use of containers as an isolation mechanisms introduces two challenges for data-intensive applications, *data access overheads* and the *container resource footprint*.

Data access overheads are caused by the stateless nature of the container-based approach, which forces state to be maintained externally, e.g. in object stores such as Amazon S3 [6], or passed between function invocations. Both options incur costs due to duplicating data in each function, repeated serialisation, and regular network transfers. This results in current applications adopting an inefficient "data-shipping architecture", i.e. moving data to the computation and not vice versa—such architectures have been abandoned by the data management community many decades ago [36]. These overheads are compounded as the number of functions increases, reducing the benefit of unlimited parallelism, which is what makes serverless computing attractive in the first place.

The container resource footprint is particularly relevant because of the high-volume and short-lived nature of serverless workloads. Despite containers having a smaller memory and CPU overhead than other mechanisms such as virtual machines (VMs), there remains an impedance mismatch between the execution of individual short-running functions and the process-based isolation of containers. Containers have start-up latencies in the hundreds of milliseconds to several seconds, leading to the *cold-start* problem in today's serverless platforms [36, 83]. The large memory footprint of containers limits scalability—while technically capped at the process limit of a machine, the maximum number of containers is usually limited by the amount of available memory, with only a few thousand containers supported on a machine with 16 GB of RAM [51].

Current data-intensive serverless applications have addressed these problems individually, but never solved both—instead, either exacerbating the container resource overhead or breaking the serverless model. Some systems avoid data movement costs by maintaining state in long-lived VMs or services, such as ExCamera [30], Shredder [92] and Cirrus [18], thus introducing non-serverless components. To address the performance overhead of containers, systems typically increase the level of trust in users' code and weaken isolation guarantees. PyWren [42] reuses containers to execute multiple functions; Crucial [12] shares a single instance of the Java virtual machine (JVM) between functions; SAND [1] executes multiple functions in long-lived containers, which also run an additional message-passing service; and Cloudburst [75] takes a similar approach, introducing a local key-value-store cache. Provisioning containers to execute multiple functions and extra services amplifies resource overheads, and breaks the fine-grained elastic scaling inherent to serverless. While several of these systems reduce data access overheads with local storage, none provide *shared memory* between functions, thus still requiring duplication of data in separate process memories.

Other systems reduce the container resource footprint by moving away from containers and VMs. Terrarium [28] and Cloudflare Workers [22] employ software-based isolation using WebAssembly and V8 Isolates, respectively; Krustlet [54] replicates containers using WebAssembly for memory safety; and SEUSS [16] demonstrates serverless unikernels. While these approaches have a reduced resource footprint, they do not address data access overheads, and the use of software-based isolation alone does not isolate resources.

We make the observation that serverless computing can better support data-intensive applications with a new isolation abstraction that (i) provides strong memory and resource isolation between functions, yet (ii) supports efficient state sharing. Data should be *co-located* with functions and accessed directly, minimising data-shipping. Furthermore, this new isolation abstraction must (iii) allow scaling state across multiple hosts; (iv) have a low memory footprint, permitting many instances on one machine; (v) exhibit fast instantiation times; and (vi) support multiple programming languages to facilitate the porting of existing applications.

In this paper, we describe **Faaslets**, a new *lightweight isolation abstraction* for data-intensive serverless computing. Faaslets support stateful functions with efficient shared memory access, and are executed by our **FAASM** distributed serverless runtime. Faaslets have the following properties, summarising our contributions:

(1) Faaslets achieve lightweight isolation. Faaslets rely on *software fault isolation* (SFI) [82], which restricts functions to accesses of their own memory. A function associated with a Faaslet, together with its library and language runtime dependencies, is compiled to WebAssembly [35]. The FAASM runtime then executes multiple Faaslets, each with a dedicated thread, within a single address space. For resource isolation, the CPU cycles of each thread are constrained using Linux *cgroups* [79] and network access is limited using *network namespaces* [79] and *traffic shaping*. Many Faaslets can be executed efficiently and safely on a single machine.

(2) Faaslets support efficient local/global state access. Since Faaslets share the same address space, they can access shared memory regions with local state efficiently. This allows the co-location of data and functions and avoids serialisation overheads. Faaslets use a two-tier state architecture, a *local* tier provides in-memory sharing, and a *global* tier supports distributed access to state across hosts. The FAASM runtime provides a state management API to Faaslets that gives fine-grained control over state in both tiers. Faaslets also support stateful applications with different consistency requirements between the two tiers.

(3) Faaslets have fast initialisation times. To reduce cold-start time when a Faaslet executes for the first time, it is launched from a suspended state. The FAASM runtime pre-initialises a Faaslet ahead-of-time and snapshots its memory to obtain a *Proto-Faaslet*, which can be restored in hundreds of microseconds. Proto-Faaslets are used to create fresh Faaslet instances quickly, e.g. avoiding the time to initialise a language runtime. While existing work on snapshots for serverless takes a single-machine approach [1, 16, 25, 61], Proto-Faaslets support cross-host restores and are OS-independent.

(4) Faaslets support a flexible host interface. Faaslets interact with the host environment through a set of POSIX-like calls for networking, file I/O, global state access and library loading/linking. This allows them to support dynamic language runtimes and facilitates the porting of existing applications, such as CPython by changing fewer than 10 lines of code. The host interface provides just enough virtualisation to ensure isolation while adding a negligible overhead.

The FAASM runtime[1] uses the LLVM compiler toolchain to translate applications to WebAssembly and supports functions written in a range of programming languages, including C/C++, Python, Typescript and Javascript. It integrates with existing serverless platforms, and we describe the use with *Knative* [33], a state-of-the-art platform based on Kubernetes.

To evaluate FAASM's performance, we consider a number of workloads and compare to a container-based serverless deployment. When training a machine learning model with SGD [68], we show that FAASM achieves a 60% improvement in run time, a 70% reduction in network transfers, and a 90% reduction in memory usage; for machine learning inference using TensorFlow Lite [78] and MobileNet [37], FAASM achieves over a 200% increase in maximum throughput, and a 90% reduction in tail latency. We also show that FAASM executes a distributed linear algebra job for matrix multiplication using Python/Numpy with negligible performance overhead and a 13% reduction in network transfers.

2 Isolation vs. Sharing in Serverless

Sharing memory is fundamentally at odds with the goal of isolation, hence providing shared access to in-memory state

[1] FAASM is open-source and available at github.com/lsds/Faasm

		Containers	VMs	Unikernel	SFI	Faaslet
Func.	Memory safety	✓	✓	✓	✓	✓
	Resource isolation	✓	✓	✓	✗	✓
	Efficient state sharing	✗	✗	✗	✗	✓
	Shared filesystem	✓	✗	✗	✓	✓
Non-func.	Initialisation time	100 ms	100 ms	10 ms	10 µs	1 ms
	Memory footprint	MBs	MBs	KBs	Bytes	KBs
	Multi-language	✓	✓	✓	✗	✓

Table 1: Isolation approaches for serverless (Initialisation times include ahead-of-time snapshot restore where applicable [16,25,61].)

in a multi-tenant serverless environment is a challenge.

Tab. 1 contrasts *containers* and *VMs* with other potential serverless isolation options, namely *unikernels* [16] in which minimal VM images are used to pack tasks densely on a hypervisor and *software-fault isolation* (SFI) [82], providing lightweight memory safety through static analysis, instrumentation and runtime traps. The table lists whether each fulfils three key functional requirements: memory safety, resource isolation and sharing of in-memory state. A fourth requirement is the ability to share a filesystem between functions, which is important for legacy code and to reduce duplication with shared files.

The table also compares these options on a set of non-functional requirements: low initialisation time for fast elasticity; small memory footprint for scalability and efficiency, and the support for a range of programming languages.

Containers offer an acceptable balance of features if one sacrifices efficient state sharing—as such they are used by many serverless platforms [32, 39, 50]. Amazon uses Firecracker [4], a "micro VM" based on KVM with similar properties to containers, e.g. initialisation times in the hundreds of milliseconds and memory overheads of megabytes.

Containers and VMs compare poorly to unikernels and SFI on initialisation times and memory footprint because of their level of virtualisation. They both provide complete virtualised POSIX environments, and VMs also virtualise hardware. Unikernels minimise their levels of virtualisation, while SFI provides none. Many unikernel implementations, however, lack the maturity required for production serverless platforms, e.g. missing the required tooling and a way for non-expert users to deploy custom images. SFI alone cannot provide resource isolation, as it purely focuses on memory safety. It also does not define a way to perform isolated interactions with the underlying host. Crucially, as with containers and VMs, neither unikernels nor SFI can share state efficiently, with no way to express shared memory regions between compartments.

2.1 Improving on Containers

Serverless functions in containers typically share state via external storage and duplicate data across function instances. Data access and serialisation introduces network and compute overheads; duplication bloats the memory footprint of containers, already of the order of megabytes [51]. Containers contribute hundreds of milliseconds up to seconds in cold-start latencies [83], incurred on initial requests and when scaling. Existing work has tried to mitigate these drawbacks by recycling containers between functions, introducing static VMs, reducing storage latency, and optimising initialisation.

Recycling containers avoids initialisation overheads and allows data caching but sacrifices isolation and multi-tenancy. PyWren [42] and its descendants, Numpywren [73], IBMPywren [69], and Locus [66] use recycled containers, with long-lived AWS Lambda functions that dynamically load and execute Python functions. Crucial [12] takes a similar approach, running multiple functions in the same JVM. SAND [1] and Cloudburst [75] provide only process isolation between functions of the same application and place them in shared long-running containers, with at least one additional background storage process. Using containers for multiple functions and supplementary long-running services requires over-provisioned memory to ensure capacity both for concurrent executions and for peak usage. This is at odds with the idea of fine-grained scaling in serverless.

Adding static VMs to handle external storage improves performance but breaks the serverless paradigm. Cirrus [18] uses large VM instances to run a custom storage back-end; Shredder [92] uses a single long-running VM for both storage and function execution; ExCamera [30] uses long-running VMs to coordinate a pool of functions. Either the user or provider must scale these VMs to match the elasticity and parallelism of functions, which adds complexity and cost.

Reducing the latency of auto-scaled storage can improve performance within the serverless paradigm. Pocket [43] provides ephemeral serverless storage; other cloud providers offer managed external state, such as AWS Step Functions [3], Azure Durable Functions [53], and IBM Composer [8]. Such approaches, however, do not address the data-shipping problem and its associated network and memory overheads.

Container initialisation times have been reduced to mitigate the cold-start problem, which can contribute several seconds of latency with standard containers [36, 72, 83]. SOCK [61] improves the container boot process to achieve cold starts in the low hundreds of milliseconds; Catalyzer [25] and SEUSS [16] demonstrate snapshot and restore in VMs and unikernels to achieve millisecond serverless cold starts. Although such reductions are promising, the resource overhead and restrictions on sharing memory in the underlying mechanisms still remain.

2.2 Potential of Software-based Isolation

Software-based isolation offers memory safety with initialisation times and memory overheads up to two orders of magnitude lower than containers and VMs. For this reason, it is an attractive starting point for serverless isolation. However, software-based isolation alone does not support resource isolation, or efficient in-memory state sharing.

It has been used in several existing edge and serverless computing systems, but none address these shortcomings.

Fastly's Terrarium [28] and Cloudflare Workers [22] provide memory safety with WebAssembly [35] and V8 Isolates [34], respectively, but neither isolates CPU or network use, and both rely on data shipping for state access; Shredder [92] also uses V8 Isolates to run code on a storage server, but does not address resource isolation, and relies on co-locating state and functions on a single host. This makes it ill-suited to the level of scale required in serverless platforms; Boucher et al. [14] show microsecond initialisation times for Rust microservices, but do not address isolation or state sharing; Krustlet [54] is a recent prototype using WebAssembly to replace Docker in Kubernetes, which could be integrated with Knative [33]. It focuses, however, on replicating container-based isolation, and so fails to meet our requirement for in-memory sharing.

Our final non-functional requirement is for multi-language support, which is not met by language-specific approaches to software-based isolation [11, 27]. Portable Native Client [23] provides multi-language software-based isolation by targeting a portable intermediate representation, LLVM IR, and hence meets this requirement. Portable Native Client has now been deprecated, with WebAssembly as its successor [35].

WebAssembly offers strong memory safety guarantees by constraining memory access to a single linear byte array, referenced with offsets from zero. This enables efficient bounds checking at both compile- and runtime, with runtime checks backed by traps. These traps (and others for referencing invalid functions) are implemented as part of WebAssembly runtimes [87]. The security guarantees of WebAssembly are well established in existing literature, which covers formal verification [84], taint tracking [31], and dynamic analysis [45]. WebAssembly offers mature support for languages with an LLVM front-end such as C, C++, C#, Go and Rust [49], while toolchains exist for Typescript [10] and Swift [77]. Java bytecode can also be converted [7], and further language support is possible by compiling language runtimes to WebAssembly, e.g. Python, JavaScript and Ruby. Although WebAssembly is restricted to a 32-bit address space, 64-bit support is in development.

The WebAssembly specification does not yet include mechanisms for sharing memory, therefore it alone cannot meet our requirements. There is a proposal to add a form of synchronised shared memory to WebAssembly [85], but it is not well suited to sharing serverless state dynamically due to the required compile-time knowledge of all shared regions. It also lacks an associated programming model and provides only local memory synchronisation.

The properties of software-based isolation highlight a compelling alternative to containers, VMs and unikernels, but none of these approaches meet all of our requirements. We therefore propose a new isolation approach to enable efficient serverless computing for big data.

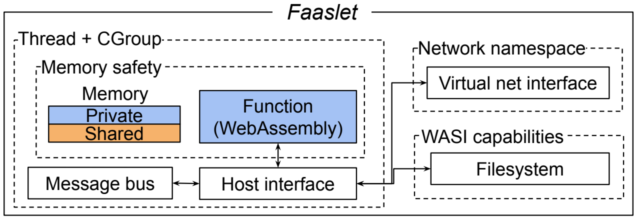


Figure 1: Faaslet abstraction with isolation

3 Faaslets

We propose *Faaslets*, a new isolation mechanism that satisfies all the requirements for efficient data-intensive serverless computing. Tab. 1 highlights Faaslets' strong memory and resource isolation guarantees, and efficient *shared in-memory state*. Faaslets provide a minimal level of lightweight virtualisation through their *host interface*, which supports serverless-specific tasks, memory management, a limited filesystem and network access.

In terms of non-functional requirements, Faaslets improve on containers and VMs by having a memory footprint below 200 KB and cold-start initialisation times of less than 10 ms. Faaslets execute functions compiled to secure IR, allowing them to support multiple programming languages.

While Faaslets cannot initialise as quickly as pure SFI, they mitigate the cold-start problem through ahead-of-time initialisation from snapshots called *Proto-Faaslets*. Proto-Faaslets reduce initialisation times to hundreds of microseconds, and a single snapshot can be restored across hosts, quickly scaling horizontally on clusters.

3.1 Overview

Fig. 1 shows a function isolated inside a Faaslet. The function itself is compiled to WebAssembly [35], guaranteeing memory safety and control flow integrity. By default, a function is placed in its own *private* contiguous memory region, but Faaslets also support *shared regions* of memory (§3.3). This allows a Faaslet to access shared in-memory state within the constraints of WebAssembly's memory safety guarantees.

Faaslets also ensure fair resource access. For CPU isolation, they use the CPU subset of Linux *cgroups* [79]. Each function is executed by a dedicated *thread* of a shared runtime process. This thread is assigned to a cgroup with a share of CPU equal to that of all Faaslets. The Linux CFS [79] ensures that these threads are scheduled with equal CPU time.

Faaslets achieve secure and fair network access using *network namespaces*, *virtual network interfaces* and *traffic shaping* [79]. Each Faaslet has its own network interface in a separate namespace, configured using *iptables* rules. To ensure fairness between co-located tenants, each Faaslet applies traffic shaping on its virtual network interface using `tc`, thus enforcing ingress and egress traffic rate limits.

As functions in a Faaslet must be permitted to invoke standard system calls to perform memory management and I/O operations, Faaslets offer an interface through which to in-

Class	Function	Action	Standard
Calls	`byte* read_call_input()`	Read input data to function as byte array	*none*
	`void write_call_output(out_data)`	Write output data for function	
	`int chain_call(name, args)`	Call function and return the `call_id`	
	`int await_call(call_id)`	Await the completion of `call_id`	
	`byte* get_call_output(call_id)`	Load the output data of `call_id`	
State	`byte* get_state(key, flags)`	Get pointer to state value for `key`	
	`byte* get_state_offset(key, off, flags)`	Get pointer to state value for `key` at offset	
	`void set_state(key, val)`	Set state value for `key`	
	`void set_state_offset(key, val, len, off)`	Set `len` bytes of state value at offset for `key`	
	`void push/pull_state(key)`	Push/pull global state value for `key`	
	`void push/pull_state_offset(key, off)`	Push/pull global state value for `key` at offset	
	`void append_state(key, val)`	Append data to state value for `key`	
	`void lock_state_read/write(key)`	Lock local copy of state value for `key`	
	`void lock_state_global_read/write(key)`	Lock state value for `key` globally	
Dynlink	`void* dlopen/dlsym(...)`	Dynamic linking of libraries	POSIX
	`int dlclose(...)`	*As above*	
Memory	`void* mmap(...)`, `int munmap(...)`	Memory grow/shrink only	
	`int brk(...)`, `void* sbrk(...)`	Memory grow/shrink	
Network	`int socket/connect/bind(...)`	Client-side networking only	WASI
	`size_t send/recv(...)`	Send/recv via virtual interface	
File I/O	`int open/close/dup/stat(...)`	Per-user virtual filesystem access	
	`size_t read/write(...)`	*As above*	
Misc	`int gettime(...)`	Per-user monotonic clock only	
	`size_t getrandom(...)`	Uses underlying host `/dev/urandom`	

Table 2: Faaslet host interface (The final column indicates whether functions are defined as part of POSIX or WASI [57].)

teract with the underlying host. Unlike containers or VMs, Faaslets do not provide a fully-virtualised POSIX environment but instead support a minimal serverless-specific host interface (see Fig. 1). Faaslets virtualise system calls that interact with the underlying host and expose a range of functionality, as described below.

The host interface integrates with the serverless runtime through a *message bus* (see Fig. 1). The message bus is used by Faaslets to communicate with their parent process and each other, receive function calls, share work, invoke and await other functions, and to be told by their parent process when to spawn and terminate.

Faaslets support a *read-global write-local* filesystem, which lets functions read files from a global object store (§5), and write to locally cached versions of the files. This is primarily used to support legacy applications, notably language runtimes such as CPython [67], which need a filesystem for loading library code and storing intermediate bytecode. The filesystem is accessible through a set of POSIX-like API functions that implement the WASI capability-based security model, which provides efficient isolation through unforgeable file handles [56]. This removes the need for more resource-intensive filesystem isolation such as a layered filesystem or `chroot`, which otherwise add to cold start latencies [61].

3.2 Host Interface

The Faaslet host interface must provide a virtualisation layer capable of executing a range of serverless big data applications, as well as legacy POSIX applications. This interface necessarily operates outside the bounds of memory safety, and hence is trusted to preserve isolation when interacting with the host.

In existing serverless platforms based on containers and VMs, this virtualisation layer is a standard POSIX environment, with serverless-specific tasks executed through language- and provider-specific APIs over HTTP [5, 32, 39]. Instantiating a full POSIX environment with the associated isolation mechanisms leads to high initialisation times [61], and heavy use of HTTP APIs contributes further latency and network overheads.

In contrast, the Faaslet host interface targets minimal virtualisation, hence reducing the overheads required to provide isolation. The host interface is a low-level API built exclusively to support a range of high-performance serverless applications. The host interface is dynamically linked with function code at runtime (§3.4), making calls to the interface more efficient than performing the same tasks through an external API.

Tab. 2 lists the Faaslet host interface API, which supports: (i) chained serverless function invocation; (ii) interacting with shared state (§4); (iii) a subset of POSIX-like calls for memory management, timing, random numbers, file/network I/O and dynamic linking. A subset of these POSIX-like calls are implemented according to WASI, an emerging standard for a server-side WebAssembly interface [57]. Some key details of the API are as follows:

Function invocation. Functions retrieve their input data serialised as byte arrays using the `read_call_input` function, and similarly write their output data as byte arrays using `write_call_output`. Byte arrays constitute a generic, language-agnostic interface.

Non-trivial serverless applications invoke multiple functions that work together as part of chained calls, made with the `chain_call` function. Users' functions have unique names, which are passed to `chain_call`, along with a byte array containing the input data for that call.

A call to `chain_call` returns the call ID of the invoked function. The call ID can then be passed to `await_call` to perform a blocking wait for another call to finish or fail, yielding its return code. The Faaslet blocks until the function has completed, and passes the same call ID to `get_call_output` to retrieve the chained call's output data.

Calls to `chain_call` and `await_call` can be used in loops to spawn and await calls in a similar manner to standard multi-threaded code: one loop invokes `chain_call` and records the call IDs; a second loop calls `await_call` on each ID in turn. We show this pattern in Python in Listing 1.

Dynamic linking. Some legacy applications and libraries require support for dynamic linking, e.g. CPython dynamically links Python extensions. All dynamically loaded code must first be compiled to WebAssembly and undergo the same validation process as other user-defined code (§3.4). Such modules are loaded via the standard Faaslet filesystem abstraction and covered by the same safety guarantees as its parent function. Faaslets support this through a standard POSIX dynamic linking API, which is implemented according to WebAssembly dynamic linking conventions [86].

Memory. Functions allocate memory dynamically through calls to `mmap()` and `brk()`, either directly or through `dlmalloc` [44]. The Faaslet allocates memory in its private memory region, and uses `mmap` on the underlying host to extend the region if necessary. Each function has its own predefined memory limit, and these calls fail if growth of the private region would exceeded this limit.

Networking. The supported subset of networking calls allows simple client-side send/receive operations and is sufficient for common use cases, such as connecting to an external data store or a remote HTTP endpoint. The functions `socket`, `connect` and `bind` allow setting up the socket while `read` and `write` allow the sending and receiving of data. Calls fail if they pass flags that are not related to simple send/receive operations over IPv4/IPv6, e.g. the `AF_UNIX` flag.

The host interface translates these calls to equivalent socket operations on the host. All calls interact exclusively with the Faaslet's virtual network interface, thus are constrained to a private network interface and cannot exceed rate limits due to the traffic shaping rules.

Byte arrays. Function inputs, results and state are represented as simple byte arrays, as is all function memory. This avoids the need to serialise and copy data as it passes through the API, and makes it trivial to share arbitrarily complex in-memory data structures.

3.3 Shared Memory Regions

As discussed in §2, sharing in-memory state while otherwise maintaining isolation is an important requirement for efficient serverless big data applications. Faaslets do this by adding the new concept of *shared regions* to the existing WebAssembly memory model [35]. Shared regions give functions concurrent access to disjoint segments of shared process memory, allowing them direct, low-latency access to shared data structures. Shared regions are backed by standard OS virtual memory, so there is no extra serialisation or overhead, hence Faaslets achieve efficient concurrent access on a par with native multi-threaded applications. In §4.2, we describe how Faaslets use this mechanism to provide shared in-memory access to global state.

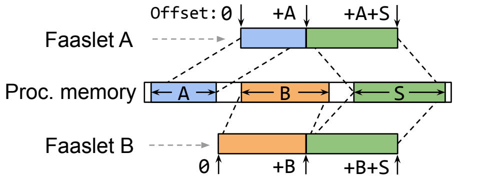


Figure 2: Faaslet shared memory region mapping

Shared regions maintain the memory safety guarantees of the existing WebAssembly memory model, and use standard OS virtual memory mechanisms. WebAssembly restricts each function's memory to a contiguous linear byte array, which is allocated by the Faaslet at runtime from a disjoint section of the process memory. To create a new shared region, the Faaslet extends the function's linear byte array, and remaps the new pages onto a designated region of common process memory. The function accesses the new region of linear memory as normal, hence maintaining memory safety, but the underlying memory accesses are mapped onto the shared region.

Fig. 2 shows Faaslets A and B accessing a shared region (labelled S), allocated from a disjoint region of the common process memory (represented by the central region). Each Faaslet has its own region of private memory (labelled A and B), also allocated from the process memory. Functions inside each Faaslet access all memory as offsets from zero, forming a single linear address space. Faaslets map these offsets onto either a private region (in this case the lower offsets), or a shared region (in this case the higher offsets).

Multiple shared regions are permitted, and functions can also extend their private memory through calls to the memory management functions in the host interface such as `brk` (§3.2). Extension of private memory and creation of new shared regions is handled by extending a byte array, which represents the function's memory, and then remapping the underlying pages to regions of shared process memory. This means the function continues to see a single densely-packed linear address space, which may be backed by several virtual memory mappings. Faaslets allocate shared process memory through calls to `mmap` on the underlying host, passing `MAP_SHARED` and `MAP_ANONYMOUS` flags to create shared and private regions, respectively, and remap these regions with `mremap`.

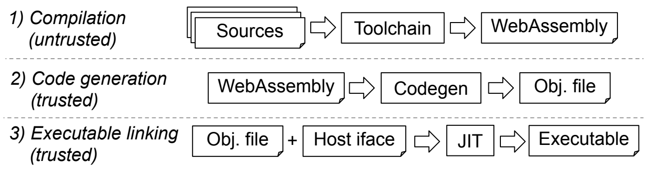


Figure 3: Creation of a Faaslet executable

3.4 Building Functions for Faaslets

Fig. 3 shows the three phases to convert source code of a function into a Faaslet executable: (1) the user invokes the Faaslet toolchain to compile the function into a WebAssembly binary, linking against a language-specific declaration of the Faaslet host interface; (2) code generation creates an object file with machine code from WebAssembly; and (3) the host interface definition is linked with the machine code to produce the Faaslet executable.

When Faaslets are deployed, the compilation phase to generate the WebAssembly binary takes place on a user's machine. Since that is untrusted, the code generation phase begins by validating the WebAssembly binary, as defined in the WebAssembly specification [35]. This ensures that the binary conforms to the specification. Code generation then takes place in a trusted environment, after the user has uploaded their function.

In the linking phase, the Faaslet uses LLVM JIT libraries [49] to link the object file and the definition of the host interface implementation. The host interface functions are defined as *thunks*, which allows injecting the trusted host interface implementation into the function binary.

Faaslets use WAVM [70] to perform the validation, code generation and linking. WAVM is an open-source WebAssembly VM, which passes the WebAssembly conformance tests [84] and thus guarantees that the resulting executable enforces memory safety and control flow integrity [35].

4 Local and Global State

Stateful serverless applications can be created with Faaslets using *distributed data objects (DDO)*, which are language-specific classes that expose a convenient high-level state interface. DDOs are implemented using the key/value state API from Tab. 2.

The state associated with Faaslets is managed using a *two-tier* approach that combines local sharing with global distribution of state: a *local tier* provides shared in-memory access to state on the same host; and a *global tier* allows Faaslets to synchronise state across hosts.

DDOs hide the two-tier state architecture, providing transparent access to distributed data. Functions, however, can still access the state API directly, either to exercise more fine-grained control over consistency and synchronisation, or to implement custom data structures.

Listing 1: Distributed SGD application with Faaslets

```
1  t_a = SparseMatrixReadOnly("training_a")
2  t_b = MatrixReadOnly("training_b")
3  weights = VectorAsync("weights")
4
5  @faasm_func
6  def weight_update(idx_a, idx_b):
7    for col_idx, col_a in t_a.columns[idx_a:idx_b]:
8      col_b = t_b.columns[col_idx]
9      adj = calc_adjustment(col_a, col_b)
10     for val_idx, val in col_a.non_nulls():
11       weights[val_idx] += val * adj
12       if iter_count % threshold == 0:
13         weights.push()
14
15 @faasm_func
16 def sgd_main(n_workers, n_epochs):
17   for e in n_epochs:
18     args = divide_problem(n_workers)
19     c = chain(update, n_workers, args)
20     await_all(c)
21     ...
```

4.1 State Programming Model

Each DDO represents a single state value, referenced throughout the system using a string holding its respective state key.

Faaslets write changes from the local to the global tier by performing a *push*, and read from the global to the local tier by performing a *pull*. DDOs may employ push and pull operations to produce variable consistency, such as delaying updates in an eventually-consistent list or set, and may lazily pull values only when they are accessed, such as in a distributed dictionary. Certain DDOs are immutable, and hence avoid repeated synchronisation.

Listing 1 shows both implicit and explicit use of two-tier state through DDOs to implement stochastic gradient descent (SGD) in Python. The `weight_update` function accesses two large input matrices through the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs (lines 1 and 2), and a single shared weights vector using `VectorAsync` (line 3). `VectorAsync` exposes a `push()` function which is used to periodically push updates from the local tier to the global tier (line 13). The calls to `weight_update` are chained in a loop in `sgd_main` (line 19).

Function `weight_update` accesses a randomly assigned subset of columns from the training matrices using the `columns` property (lines 7 and 8). The DDO implicitly performs a pull operation to ensure that data is present, and only replicates the necessary subsets of the state values in the local tier—the entire matrix is not transferred unnecessarily.

Updates to the shared weights vector in the local tier are made in a loop in the `weight_update` function (line 11). It invokes the `push` method on this vector (line 13) sporadically to update the global tier. This improves performance and reduces network overhead, but introduces inconsistency between the tiers. SGD tolerates such inconsistencies and it does not affect the overall result.

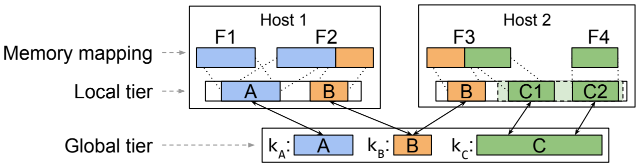


Figure 4: Faaslet two-tier state architecture

4.2 Two-Tier State Architecture

Faaslets represent state with a key/value abstraction, using unique *state keys* to reference *state values*. The authoritative state value for each key is held in the global tier, which is backed by a distributed key-value store (KVS) and accessible to all Faaslets in the cluster. Faaslets on a given host share a local tier, containing replicas of each state value currently mapped to Faaslets on that host. The local tier is held exclusively in Faaslet shared memory regions, and Faaslets do not have a separate local storage service, as in SAND [1] or Cloudburst [75].

Fig. 4 shows the two-tier state architecture across two hosts. Faaslets on host 1 share state value A; Faaslets on both hosts share state value B. Accordingly, there is a replica of state value A in the local tier of host 1, and replicas of state value B in the local tier of both hosts.

The `columns` method of the `SparseMatrixReadOnly` and `MatrixReadOnly` DDOs in Listing 1 uses *state chunks* to access a subset of a larger state value. As shown in Fig. 4, state value C has state chunks, which are treated as smaller independent state values. Faaslets create replicas of only the required chunks in their local tier.

Ensuring local consistency. State value replicas in the local tier are created using Faaslet shared memory (§3.3). To ensure consistency between Faaslets accessing a replica, Faaslets acquire a *local read lock* when reading, and a *local write lock* when writing. This locking happens implicitly as part of all state API functions, but not when functions write directly to the local replica via a pointer. The state API exposes the `lock_state_read` and `lock_state_write` functions that can be used to acquire local locks explicitly, e.g. to implement a list that performs multiple writes to its state value when atomically adding an element. A Faaslet creates a new local replica after a call to `pull_state` or `get_state` if it does not already exist, and ensures consistency through a write lock.

Ensuring global consistency. DDOs can produce varying levels of consistency between the tiers as shown by `VectorAsync` in Listing 1. To enforce strong consistency, DDOs must use *global read/write locks*, which can be acquired and released for each state key using `lock_state_global_read` and `lock_state_global_write`, respectively. To perform a consistent write to the global tier, an object acquires a global write lock, calls `pull_state` to update the local tier, applies its write to the local tier, calls `push_state` to update the global tier, and releases the lock.

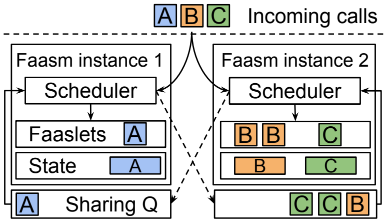


Figure 5: FAASM system architecture

5 FAASM Runtime

FAASM is the serverless runtime that uses Faaslets to execute distributed stateful serverless applications across a cluster. FAASM is designed to integrate with existing serverless platforms, which provide the underlying infrastructure, auto-scaling functionality and user-facing frontends. FAASM handles the scheduling, execution and state management of Faaslets. The design of FAASM follows a distributed architecture: multiple FAASM runtime instances execute on a set of servers, and each instance manages a pool of Faaslets.

5.1 Distributed Scheduling

A local scheduler in the FAASM runtime is responsible for the scheduling of Faaslets. Its scheduling strategy is key to minimising data-shipping (see §2) by ensuring that executed functions are co-located with required in-memory state. One or more Faaslets managed by a runtime instance may be *warm*, i.e. they already have their code and state loaded. The scheduling goal is to ensure that as many function calls as possible are executed by warm Faaslets.

To achieve this without modifications to the underlying platform's scheduler, FAASM uses a *distributed shared state* scheduler similar to Omega [71]. Function calls are sent round-robin to local schedulers, which execute the function locally if they are warm and have capacity, or share it with another warm host if one exists. The set of warm hosts for each function is held in the FAASM state global tier, and each scheduler may query and atomically update this set during the scheduling decision.

Fig. 5 shows two FAASM runtime instances, each with its own local scheduler, a pool of Faaslets, a collection of state stored in memory, and a sharing queue. Calls for functions A–C are received by the local schedulers, which execute them locally if they have warm Faaslets, and share them with the other host if not. Instance 1 has a warm Faaslet for function A and accepts calls to this function, while sharing calls to functions B and C with Instance 2, which has corresponding warm Faaslets. If a function call is received and there are no instances with warm Faaslets, the instance that received the call creates a new Faaslet, incurring a "cold start".

5.2 Reducing Cold Start Latency

While Faaslets typically initialise in under 10 ms, FAASM reduces this further using *Proto-Faaslets*, which are Faaslets that contain snapshots of arbitrary execution state that can be restored on any host in the cluster. From this snapshot, FAASM spawns a new Faaslet instance, typically reducing initialisation to hundreds of microseconds (§6.5).

Different Proto-Faaslets are generated for a function by specifying user-defined *initialisation code*, which is executed before snapshotting. If a function executes the same code on each invocation, that code can become initialisation code and be removed from the function itself. For Faaslets with dynamic language runtimes, the runtime initialisation can be done as part of the initialisation code.

A Proto-Faaslet snapshot includes a function's stack, heap, function table, stack pointer and data, as defined in the WebAssembly specification [35]. Since WebAssembly memory is represented by a contiguous byte array, containing the stack,heap and data, FAASM restores a snapshot into a new Faaslet using a copy-on-write memory mapping. All other data is held in standard C++ objects. Since the snapshot is independent of the underlying OS thread or process, FAASM can serialise Proto-Faaslets and instantiate them across hosts.

FAASM provides an *upload* service that exposes an HTTP endpoint. Users upload WebAssembly binaries to this endpoint, which then performs code generation (§3.4) and writes the resulting object files to a shared *object store*. The implementation of this store is specific to the underlying serverless platform but can be a cloud provider's own solution such as AWS S3 [6]. Proto-Faaslets are generated and stored in the FAASM global state tier as part of this process. When a Faaslet undergoes a cold start, it loads the object file and Proto-Faaslet, and restores it.

In addition, FAASM uses Proto-Faaslets to reset Faaslets after each function call. Since the Proto-Faaslet captures a function's initialised execution state, restoring it guarantees that no information from the previous call is disclosed. This can be used for functions that are *multi-tenant*, e.g. in a serverless web application. FAASM guarantees that private data held in memory is cleared away after each function execution, thereby allowing Faaslets to handle subsequent calls across tenants. In a container-based platform, this is typically not safe, as the platform cannot ensure that the container memory has been cleaned entirely between calls.

6 Evaluation

Our experimental evaluation targets the following questions: (i) how does FAASM state management improve efficiency and performance on parallel machine learning training? (§6.2) (ii) how do Proto-Faaslets and low initialisation times impact performance and throughput in inference serving? (§6.3) (iii) how does Faaslet isolation affect performance in a linear algebra benchmark using a dynamic language runtime? (§6.4) and (iv) how do the overheads of Faaslets compare to Docker containers? (§6.5)

6.1 Experimental Set-up

Serverless baseline. To benchmark FAASM against a state-of-the-art serverless platform, we use Knative [33], a container-based system built on Kubernetes [80]. All experiments are implemented using the same code for both FAASM and Knative, with a Knative-specific implementation of the Faaslet host interface for container-based code. This interface uses the same undelrying state management code as FAASM, but cannot share the local tier between co-located functions. Knative function chaining is performed through the standard Knative API. Redis is used for the distributed KVS and deployed to the same cluster.

FAASM integration. We integrate FAASM with Knative by running FAASM runtime instances as Knative functions that are replicated using the default autoscaler. The system is otherwise unmodified, using the default endpoints and scheduler.

Testbed. Both FAASM and Knative applications are executed on the same Kubernetes cluster, running on 20 hosts, all Intel Xeon E3-1220 3.1 GHz machines with 16 GB of RAM, connected with a 1 Gbps connection. Experiments in §6.5 were run on a single Intel Xeon E5-2660 2.6 GHz machine with 32 GB of RAM.

Metrics. In addition to the usual evaluation metrics, such as execution time, throughput and latency, we also consider *billable memory*, which quantifies memory consumption over time. It is the product of the peak function memory multiplied by the number and runtime of functions, in units of GB-seconds. It is used to attribute memory usage in many serverless platforms [5, 32, 39]. Note that all memory measurements include the containers/Faaslets and their state.

6.2 Machine Learning Training

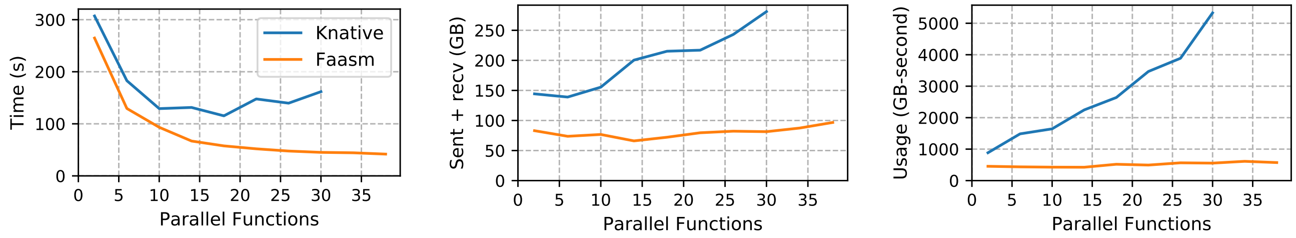
This experiment focuses on the impact of FAASM's state management on runtime, network overheads and memory usage.

We use distributed *stochastic gradient descent* (SGD) using the HOGWILD! algorithm [68] to run text classification on the Reuters RCV1 dataset [46]. This updates a central weights vector in parallel with batches of functions across multiple epochs. We run both Knative and FAASM with increasing numbers of parallel functions.

Fig. 6a shows the training time. FAASM exhibits a small improvement in runtime of 10% compared to Knative at low parallelism and a 60% improvement with 15 parallel functions. With more than 20 parallel Knative functions, the underlying hosts experience increased memory pressure and they exhaust memory with over 30 functions. Training time continues to improve for FAASM up to 38 parallel functions, at which point there is a more than an 80% improvement over 2 functions.

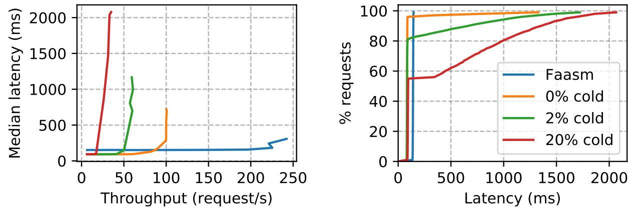
Fig. 6b shows that, with increasing parallelism, the volume of network transfers increases in both FAASM and Knative. Knative transfers more data to start with and the volume increase more rapidly, with 145 GB transferred with 2 parallel functions and 280 GB transferred with 30 functions. FAASM transfers 75 GB with 2 parallel functions and 100 GB with 38 parallel functions.

Fig. 6c shows that billable memory in Knative increases with more parallelism: from 1,000 GB-seconds for 2 func-



(a) Training time **(b) Network transfers** **(c) Memory usage**

Figure 6: Machine learning training with SGD with Faaslets (FAASM) and containers (Knative)



(a) Throughput vs. latency **(b) Latency CDF**

Figure 7: Machine learning inference with TensorFlow Lite

tions to over 5,000 GB-second for 30 functions. The billable memory for FAASM increases slowly from 350 GB-second for 2 functions to 500 GB-second with 38 functions.

The increased network transfer, memory usage and duration in Knative is caused primarily by data shipping, e.g. loading data into containers. FAASM benefits from sharing data through its local tier, hence amortises overheads and reduces latency. Further improvements in duration and network overhead come from differences in the updates to the shared weights vector: in FAASM, the updates from multiple functions are batched per host; whereas in Knative, each function must write directly to external storage. Billable memory in Knative and FAASM increases with more parallelism, however, the increased memory footprint and duration in Knative make this increase more pronounced.

To isolate the underlying performance and resource overheads of FAASM and Knative, we run the same experiment with the number of training examples reduced from 800K to 128. Across 32 parallel functions, we observe for FAASM and Knative: training times of 460 ms and 630 ms; network transfers of 19 MB and 48 MB; billable memory usage of 0.01 GB-second and 0.04 GB-second, respectively.

In this case, increased duration in Knative is caused by the latency and volume of inter-function communication through the Knative HTTP API versus direct inter-Faaslet communication. FAASM incurs reduced network transfers versus Knative as in the first experiment, but the overhead of these transfers in both systems are negligible as they are small and amortized across all functions. Billable memory is increased in Knative due to the memory overhead of each function container being 8 MB (versus 270 kB for each Faaslet). These improvements are negligible when compared with those derived from reduced data shipping and duplication of the full dataset.

6.3 Machine Learning Inference

This experiment explores the impact of the Faaslet initialisation times on cold-starts and function call throughput.

We consider a machine learning inference application because they are typically user-facing, thus latency-sensitive, and must serve high volumes of requests. We perform inference serving with TensorFlow Lite [78], with images loaded from a file server and classified using a pre-trained MobileNet [37] model. In our implementation, requests from each user are sent to different instances of the underlying serverless function. Therefore, each user sees a cold-start on their first request. We measure the latency distribution and change in median latency when increasing throughput and varying the ratio of cold-starts.

Figs. 7a and 7b show a single line for FAASM that covers all cold-start ratios. Cold-starts only introduce a negligible latency penalty of less than 1 ms and do not add significant resource contention, hence all ratios behave the same. Optimal latency in FAASM is higher than that in Knative, as the inference calculation takes longer due to the performance overhead from compiling TensorFlow Lite to WebAssembly.

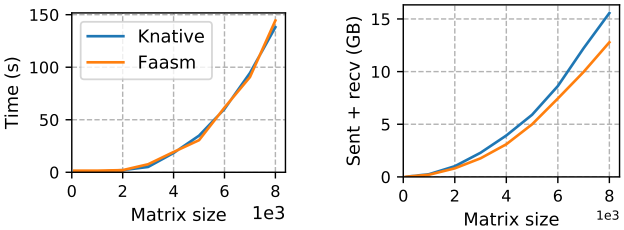
Fig. 7a shows that the median latency in Knative increases sharply from a certain throughput threshold depending on the cold-start ratio. This is caused by cold starts resulting in queuing and resource contention, with the median latency for the 20% cold-start workload increasing from 90 ms to over 2 s at around 20 req/s. FAASM maintains a median latency of 120 ms at a throughput of over 200 req/s.

Fig. 7b shows the latency distribution for a single function that handles successive calls with different cold-start ratios. Knative has a tail latency of over 2 s and more than 35% of calls have latencies of over 500 ms with 20% cold-starts. FAASM achieves a tail latency of under 150 ms for all ratios.

6.4 Language Runtime Performance with Python

The next two experiments (i) measure the performance impact of Faaslet isolation on a distributed benchmark using an existing dynamic language runtime, the CPython interpreter; and (ii) investigate the impact on a single Faaslet running compute microbenchmarks and a suite of Python microbenchmarks.

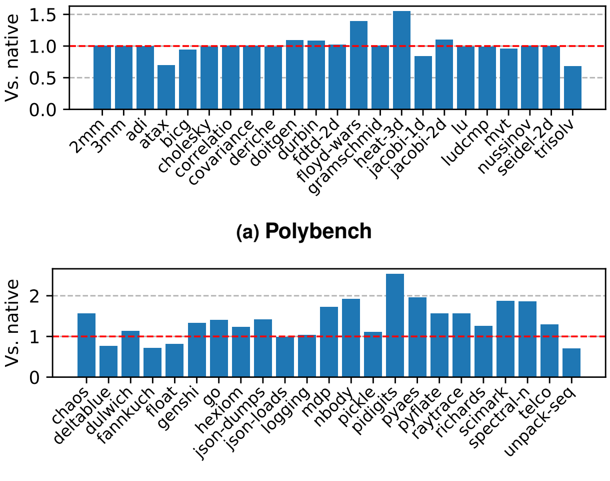
We consider a distributed divide-and-conquer matrix multiplication implemented with Python and Numpy. In the FAASM implementation, these functions are executed using



(a) **Duration**

(b) **Network transfer**

Figure 8: Comparison of matrix multiplication with Numpy



(a) **Polybench**

(b) **Python Performance Benchmark**

Figure 9: Performance of Faaslets with Python

CPython inside a Faaslet; in Knative, we use standard Python. As there is no WebAssembly support for BLAS and LAPACK, we do not use them in either implementation.

While this experiment is computationally intensive, it also makes use of the filesystem, dynamic linking, function chaining and state, thus exercising all of the Faaslet host interface. Each matrix multiplication is subdivided into multiplications of smaller submatrices and merged. This is implemented by recursively chaining serverless functions, with each multiplication using 64 multiplication functions and 9 merging functions. We compare the execution time and network traffic when running multiplications of increasingly large matrices.

Fig. 8a shows that the duration of matrix multiplications on FAASM and Knative are almost identical with increasing matrix sizes. Both take around 500 ms with 100×100 matrices, and almost 150 secs with 8000×8000 matrices. Fig. 8b shows that FAASM results in 13% less network traffic across all matrix sizes, and hence gains a small benefit from storing intermediate results more efficiently.

In the next experiment, we use Polybench/C [64] to measure the Faaslet performance overheads on simple compute functions, and the Python Performance Benchmarks [76] for overheads on more complex applications. Polybench/C is compiled directly to WebAssembly and executed in Faaslets; the Python code executes with CPython running in a Faaslet.

	Docker	Faaslets	Proto-Faaslets	vs. Docker
Initialisation	2.8 s	5.2 ms	0.5 ms	**5.6K**×
CPU cycles	251M	1.4K	650	**385K**×
PSS memory	1.3 MB	200 KB	90 KB	**15**×
RSS memory	5.0 MB	200 KB	90 KB	**57**×
Capacity	~8 K	~70 K	>100 K	**12**×

Table 3: Comparison of Faaslets vs. container cold starts (no-op function)

Fig. 9 shows the performance overhead when running both sets of benchmarks compared to native execution. All but two of the Polybench benchmarks are comparable to native with some showing performance gains. Two experience a 40%–55% overhead, both of which benefit from loop optimisations that are lost through compilation to WebAssembly. Although many of the Python benchmarks are within a 25% overhead or better, some see a 50%–60% overhead, with `pidigits` showing a 240% overhead. `pidigits` stresses big integer arithmetic, which incurs significant overhead in 32-bit WebAssembly.

Jangda et al. [41] report that code compiled to WebAssembly has more instructions, branches and cache misses, and that these overheads are compounded on larger applications. Serverless functions, however, typically are not complex applications and operate in a distributed setting in which distribution overheads dominate. As shown in Fig. 8a, FAASM can achieve competitive performance with native execution, even for functions interpreted by a dynamic language runtime.

6.5 Efficiency of Faaslets vs. Containers

Finally we focus on the difference in footprint and cold-start initialisation latency between Faaslets and containers.

To measure memory usage, we deploy increasing numbers of parallel functions on a host and measure the change in footprint with each extra function. Containers are built from the same minimal image (`alpine:3.10.1`) so can access the same local copies of shared libraries. To highlight the impact of this sharing, we include the proportional set size (PSS) and resident set size (RSS) memory consumption. Initialisation times and CPU cycles are measured across repeated executions of a no-op function. We observe the capacity as the maximum number of concurrent running containers or Faaslets that a host can sustain before running out of memory.

Tab. 3 shows several orders of magnitude improvement in CPU cycles and time elapsed when isolating a no-op with Faaslets, and a further order of magnitude using Proto-Faaslets. With an optimistic PSS memory measurement for containers, memory footprints are almost seven times lower using Faaslets, and 15× lower using Proto-Faaslets. A single host can support up to 10× more Faaslets than containers, growing to twelve times more using Proto-Faaslets.

To assess the impact of restoring a non-trivial Proto-Faaslet snapshot, we run the same initialisation time measurement for a Python no-op function. The Proto-Faaslet snapshot is a pre-initialised CPython interpreter, and the container uses a minimal `python:3.7-alpine` image. The container

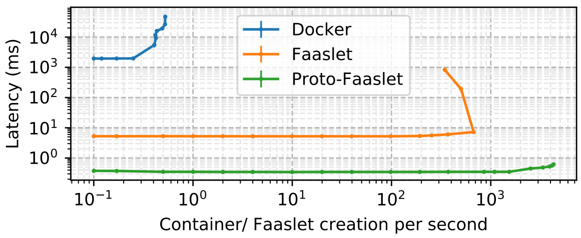


Figure 10: Function churn for Faaslets vs. containers

initialises in 3.2 s and the Proto-Faaslet restores in 0.9 ms, demonstrating a similar improvement of several orders of magnitude.

To further investigate cold-start initialisation times, we measure the time to create a new container/Faaslet at increasingly higher rates of cold-starts per second. We also measure this time when restoring the Faaslet from a Proto-Faaslet. The experiment executes on a single host, with the containers using the same minimal image.

Fig. 10 shows that both Faaslets and containers maintain a steady initialisation latency at throughputs below 3 execution/s, with Docker containers initialising in ~2 s and Faaslets in ~5 ms (or ~0.5 ms when restored from a Proto-Faaslet). As we increase the churn in Docker past 3 execution/s, initialisation times begin to increase with no gain in throughput. A similar limit for Faaslets is reached at around 600 execution/s, which grows to around 4000 execution/s with Proto-Faaslets.

We conclude that Faaslets offer a more efficient and performant form of serverless isolation than Docker containers, which is further improved with Proto-Faaslets. The lower resource footprint and initialisation times of Faaslets are important in a serverless context. Lower resource footprints reduce costs for the cloud provider and allow a higher packing density of parallel functions on a given host. Low initialisation times reduce cost and latency for the user, through their mitigation of the cold-start problem.

7 Related Work

Isolation mechanisms. Shreds [20] and Wedge [13] introduce new OS-level primitives for memory isolation, but focus on intra-process isolation rather than a complete executable as Faaslets do. Light-weight Contexts [48] and Picoprocesses [38] offer lightweight sandboxing of complete POSIX applications, but do not offer efficient shared state.

Common runtimes. Truffle [90] and GraalVM [26] are runtimes for language-independent bytecode; the JVM also executes multiple languages compiled to Java bytecode [21]. Despite compelling multi-language support, none offer multi-tenancy or resource isolation. GraalVM has recently added support for WebAssembly and could be adapted for Faaslets.

Autoscaling storage. FAASM's global state tier is currently implemented with a distributed Redis instance scaled by Kubernetes horizontal pod autoscaler [81]. Although this has not been a bottleneck, better alternatives exist: Anna [89] is a distributed KVS that achieves lower latency and more granular autoscaling than Redis; Tuba [9] provides an autoscaling KVS that operates within application-defined constraints; and Pocket [43] is a granular autoscaled storage system built specifically for a serverless environments. Crucial [12] uses Infinispan [52] to build its distributed object storage, which could also be used to implement FAASM's global state tier.

Distributed shared memory (DSM). FaRM [24] and RAMCloud [63] demonstrate that fast networks can overcome the historically poor performance of DSM systems [19], while DAL [60] demonstrates the benefits of introducing locality awareness to DSM. FAASM's global tier could be replaced with DSM to form a distributed object store, which would require a suitable consensus protocol, such as Raft [62], and a communication layer, such as Apache Arrow [65].

State in distributed dataflows. Spark [91] and Hadoop [74] support stateful distributed computation. Although focuses on fixed-size clusters and not fine-grained elastic scaling or multi-tenancy, distributed dataflow systems such as Naiad [58], SDGs [29] and CIEL [59] provide high-level interfaces for distributed state, with similar aims to those of distributed data objects—they could be implemented in or ported to FAASM. Bloom [2] provides a high-level distributed programming language, focused particularly on flexible consistency and replication, ideas also relevant to FAASM.

Actor frameworks. Actor-based systems such as Orleans [15], Akka [47] and Ray [55] support distributed stateful tasks, freeing users from scheduling and state management, much like FAASM. However, they enforce a strict asynchronous programming model and are tied to a specific languages or language runtimes, without multi-tenancy.

8 Conclusions

To meet the increasing demand for serverless big data, we presented FAASM, a runtime that delivers high-performance efficient state without compromising isolation. FAASM executes functions inside Faaslets, which provide memory safety and resource fairness, yet can share in-memory state. Faaslets are initialised quickly thanks to Proto-Faaslet snapshots. Users build stateful serverless applications with distributed data objects on top of the Faaslet state API. FAASM's two-tier state architecture co-locates functions with required state, providing parallel in-memory processing yet scaling across hosts. The Faaslet host interface also supports dynamic language runtimes and traditional POSIX applications.

Acknowledgements. We thank the anonymous reviewers and our shepherd, Don Porter, for their valuable feedback. This work was partially supported by the European Union's Horizon 2020 Framework Programme under grant agreement 825184 (CloudButton), the UK Engineering and Physical Sciences Research Council (EPSRC) award 1973141, and a gift from Intel Corporation under the TFaaS project.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[3] Amazon. AWS Step Functions. https://aws.amazon.com/step-functions/, 2020.

[4] Amazon. Firecracker Micro VM. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/, 2020.

[5] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2020.

[6] Amazon Web Services. AWS S3. https://aws.amazon.com/s3/, 2020.

[7] Alexey Andreev. TeaVM. http://www.teavm.org/, 2020.

[8] Apache Project. Openwhisk Composer. https://github.com/ibm-functions/composer, 2020.

[9] Masoud Saeida Ardekani and Douglas B Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] Assemblyscript. AssemblyScript. https://github.com/AssemblyScript/assemblyscript, 2020.

[11] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. *ACM SIGOPS Operating Systems Review*, 2017.

[12] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *ACM/IFIP Middleware Conference*, 2019.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[14] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "Micro" Back in Microservice. *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[15] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[16] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *ACM European Conference on Computer Systems (EuroSys)*, 2020.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. A Case for Serverless Machine Learning. *Systems for ML*, 2018.

[18] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus. In *ACM Symposium on Cloud Computing (SOCC)*, 2019.

[19] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. *ACM SIGOPS Operating Systems Review*, 1991.

[20] Y Chen, S Reymondjohnson, Z Sun, and L Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[21] Shigeru Chiba and Muga Nishizawa. An Easy-to-use Toolkit for Efficient Java Bytecode Translators. In *International Conference on Generative Programming and Component Engineering*, 2003.

[22] Cloudflare. Cloudflare Workers. https://workers.cloudflare.com/, 2020.

[23] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Pnacl: Portable Native Client Executables. *Google White Paper*, 2010.

[24] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[26] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[27] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *ACM SIGOPS Operating Systems Review*, 2006.

[28] Fastly. Terrarium. https://wasm.fastlylabs.com/, 2020.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit For Imperative Big Data Processing. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[30] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[31] William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *arXiv preprint arXiv:1802.01050*, 2018.

[32] Google. Google Cloud Functions. https://cloud.google.com/functions/, 2020.

[33] Google. KNative Github. https://github.com/knative, 2020.

[34] Google. V8 Engine. https://github.com/v8/v8, 2020.

[35] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[36] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[37] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[38] Jon Howell, Bryan Parno, and John R Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In *USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[39] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions, 2020.

[40] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving Deep Learning Models in a Serverless Platform. In *IEEE International Conference on Cloud Engineering, (IC2E)*, 2018.

[41] Abhinav Jangda, Bobby Powers, Emery Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *ACM Symposium on Cloud Computing (SOCC)*, 2017.

[43] Ana Klimovic, Yawen Wang, Stanford University, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[44] Doug Lea. dlmalloc. http://gee.cs.oswego.edu/dl/html/malloc.html, 2020.

[45] Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[46] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research*, 2004.

[47] Lightbend. Akka Framework. https://akka.io/, 2020.

[48] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[49] LLVM Project. LLVM 9 Release Notes. https://releases.llvm.org/9.0.0/docs/ReleaseNotes.html, 2020.

[50] Sahil Malik. Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2020.

[51] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[52] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[53] Microsoft. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable-functions-overview, 2020.

[54] Microsoft Research. Krustlet. https://deislabs.io/posts/introducing-krustlet/.

[55] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2017.

[56] Mozilla. WASI Design Principles. https://github.com/WebAssembly/WASI/blob/master/docs/DesignPrinciples.md, 2020.

[57] Mozilla. WASI: WebAssembly System Interface. https://wasi.dev/, 2020.

[58] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[59] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a Universal Execution Engine for Distributed Dataflow Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[60] Gábor Németh, Dániel Géhberger, and Péter Mátray. DAL: A Locality-Optimizing Distributed Shared Memory System. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2017.

[61] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[62] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[63] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 2015.

[64] Louis-Noel Pouchet. Polybench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2020.

[65] The Apache Project. Apache arrow. https://arrow.apache.org/.

[66] Qifan Pu, U C Berkeley, Shivaram Venkataraman, Ion Stoica, U C Berkeley, and Implementation Nsdi. Shuffling, fast and slow : Scalable analytics on serverless infrastructure. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[67] Python Software Foundation. CPython. https://github.com/python/cpython, 2020.

[68] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, 2011.

[69] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless Data Analytics in the IBM Cloud. In *ACM/IFIP Middleware Conference*, 2018.

[70] Andrew Scheidecker. WAVM. https://github.com/WAVM/WAVM, 2020.

[71] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[72] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-service Computing. In *Annual International Symposium on Microarchitecture (MICRO)*, 2019.

[73] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless Linear Algebra. *arXivpreprint arXiv:1810.09679*, 2018.

[74] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, and Others. The Hadoop Distributed File System. In *Conference on Massive Storage Systems and Technology (MSST)*, 2010.

[75] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *arXiv preprint arXiv:2001.04592*, 2020.

[76] Victor Stinner. The Python Performance Benchmark Suite. https://pyperformance.readthedocs.io/, 2020.

[77] SwiftWasm. SwiftWasm. https://swiftwasm.org/.

[78] Tensorflow. TensorFlow Lite. https://www.tensorflow.org/lite, 2020.

[79] The Kernel Development Community. The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.10/driver-api/80211/mac80211.html, 2020.

[80] The Linux Foundation. Kubernetes. https://kubernetes.io/, 2020.

[81] The Linux Foundation. Kubernetes Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2020.

[82] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[83] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[84] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

[85] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2019.

[86] WebAssembly. WebAssembly Dynamic Linking. https://webassembly.org/docs/dynamic-linking/, 2020.

[87] WebAssembly. WebAssembly Specification. https://github.com/WebAssembly/spec/, 2020.

[88] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai. Serverless Big Data Processing Using Matrix Multiplication. In *IEEE Conference on Big Data (Big Data)*, 2018.

[89] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: a KVS for any Scale. *IEEE International Conference on Data Engineering, (ICDE)*, 2018.

[90] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2013.

[91] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[92] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *ACM Symposium on Cloud Computing (SOCC)*, 2019.

Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

Redha Gouicem, Damien Carver
Sorbonne University, LIP6, Inria

Jean-Pierre Lozi
Oracle Labs

Julien Sopena
Sorbonne University, LIP6, Inria

Baptiste Lepers, Willy Zwaenepoel
University of Sydney

Nicolas Palix
Université Grenoble Alpes

Julia Lawall, Gilles Muller
Inria, Sorbonne University, LIP6

Abstract

In modern server CPUs, individual cores can run at different frequencies, which allows for fine-grained control of the performance/energy tradeoff. Adjusting the frequency, however, incurs a high latency. We find that this can lead to a problem of *frequency inversion*, whereby the Linux scheduler places a newly active thread on an idle core that takes dozens to hundreds of milliseconds to reach a high frequency, just before another core already running at a high frequency becomes idle.

In this paper, we first illustrate the significant performance overhead of repeated frequency inversion through a case study of scheduler behavior during the compilation of the Linux kernel on an 80-core Intel® Xeon-based machine. Following this, we propose two strategies to reduce the likelihood of frequency inversion in the Linux scheduler. When benchmarked over 60 diverse applications on the Intel® Xeon, the better performing strategy, S_{move}, improves performance by more than 5% (at most 56% with no energy overhead) for 23 applications, and worsens performance by more than 5% (at most 8%) for only 3 applications. On a 4-core AMD Ryzen we obtain performance improvements up to 56%.

1 Introduction

Striking a balance between performance and energy consumption has long been a battle in the development of computing systems. For several decades, CPUs have supported Dynamic Frequency Scaling (DFS), allowing the hardware or the software to update the CPU frequency at runtime. Reducing CPU frequency can reduce energy usage, but may also decrease overall performance. Still, reduced performance may be acceptable for tasks that are often idle or are not very urgent, making it desirable to save energy by reducing the frequency in many use cases. While on the first multi-core machines, all cores of a CPU had to run at the same frequency, recent server CPUs from Intel® and AMD® make it possible to update the frequency of individual cores. This feature allows for much finer-grained control, but also raises new challenges.

One source of challenges in managing core frequencies is the *Frequency Transition Latency* (*FTL*). Indeed, transitioning a core from a low to a high frequency, or conversely, has an FTL of dozens to hundreds of milliseconds. FTL leads to a problem of *frequency inversion* in scenarios that are typical of the use of the standard POSIX `fork()` and `wait()` system calls on process creation, or of synchronization between lightweight threads in a producer-consumer application. The problem occurs as follows. First, a task T_{waker} running on core C_{waker} creates or unblocks a task T_{woken}. If the *Completely Fair Scheduler* (*CFS*), *i.e.*, the default scheduler in Linux, finds an idle core C_{CFS}, it will place T_{woken} on it. Shortly thereafter, T_{waker} terminates or blocks, because *e.g.*, it was a parent process that `fork`ed a child process and `wait`ed just afterwards, or because it was a thread that was done producing data and woke up a consumer thread as its last action before going to sleep. Now C_{waker} is idle and yet executing at a high frequency because it was running T_{waker} until recently, and C_{CFS}, on which T_{woken} is running, is likely to be executing at a low frequency because it was previously idle. Consequently, the frequencies at which C_{waker} and C_{CFS} operate are *inverted* as compared to the load on the cores. This frequency inversion will not be resolved until C_{waker} reaches a low frequency and C_{CFS} reaches a high frequency, *i.e.*, for the duration of the FTL. Current hardware and software DFS policies, including the `schedutil` policy [9] that was recently added to CFS cannot prevent frequency inversion as their only decisions consist in updating core frequencies, thus paying the FTL each time. Frequency inversion reduces performance and may increase energy usage.

In this paper, we first exhibit the problem of frequency inversion in a real-world scenario through a case study of the behavior of CFS when building the Linux kernel on a Intel® Xeon-based machine with 80 cores (160 hardware threads). Our case study finds repeated frequency inversions when processes are created through the `fork()` and `wait()` system calls, and our profiling traces make it clear that frequency inversion leads to tasks running on low frequency cores for a significant part of their execution.

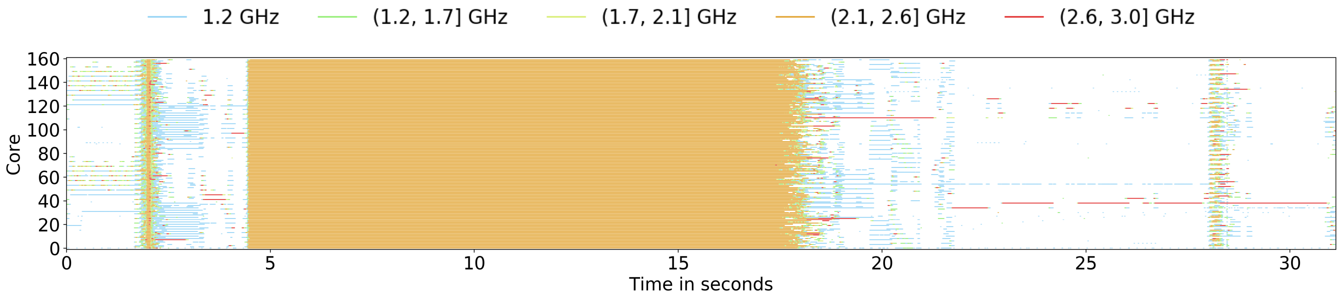


Figure 1: Execution trace when building the Linux kernel version 5.4 using 320 jobs.

Based on the results of the case study, we propose to address frequency inversion at the scheduler level. Our key observation is that the scheduler can avoid frequency inversion by taking core frequencies into account when placing a task on a core. For this, we propose and analyze two strategies. Our first strategy S_{local} is for the scheduler to simply place T_{woken} on C_{waker}, as frequency inversion involves a core C_{waker} that is likely at a high frequency, and may soon be idle. This strategy improves the kernel build performance. It runs the risk, however, that T_{waker} does not promptly terminate or block, causing a long wait before T_{woken} is scheduled. Accordingly, our second strategy S_{move} additionally arms a high-resolution timer when it places T_{woken} on C_{waker}, and if the timer expires before T_{woken} is scheduled, then T_{woken} is migrated to C_{CFS}, *i.e.*, the core CFS originally chose for it. Furthermore, even slightly delaying T_{woken} by placing it on C_{waker} is not worthwhile when C_{CFS} is above the minimum frequency. Thus, S_{move} first checks whether the frequency of C_{CFS} is above the minimum, and if so places T_{woken} on C_{CFS} directly.

The contributions of this paper are the following.

- The identification of the *frequency inversion* phenomenon, which leads to some idle cores running at a high frequency while some busy cores run at a low frequency for a significant amount of time.
- A case study, building the Linux kernel on an 80-core server, with independent per-core frequencies.
- Two strategies, S_{local} and S_{move}, to prevent frequency inversion in CFS. Implementing these policies only required minor code changes: 3 (resp. 124) lines were modified in the Linux kernel to implement S_{local} (resp. S_{move}).
- A comprehensive evaluation of our strategies on 60 diverse applications, including popular Linux benchmarks as well as applications from the Phoronix [23] and NAS [5] benchmark suites. The evaluation considers both the `powersave` CPU governor, which is currently used by default in Linux, and the experimental `schedutil` governor. It also considers two machines: a large 80-core Intel® Xeon E7-8870 v4 server and a smaller 4-core AMD® Ryzen 5 3400G desktop machine.

With the `powersave` governor on the server machine, we find that both S_{local} and S_{move} perform well overall: out of the 60 applications used in the evaluation, S_{local} and S_{move} improve the performance of 27 and 23 applications by more than 5% respectively, and worsen the performance of only 3 applications by more than 5%. In the best case, S_{local} and S_{move} improve application performance by 58% and 56% respectively with no energy overhead. However, S_{local} performs very poorly with two of the applications, even worsening performance by 80% in the worst case, which may not be acceptable for a general-purpose scheduler. S_{move} performs much better in the worst case: the increase in application execution time is only 8% and mitigated by a 9% improvement in terms of energy usage. Evaluation results with `schedutil` show that this governor does not address the frequency inversion issue, and exhibits several more cases in which S_{local} performs very poorly—while S_{move} again has much better worst-case performance. The evaluation on the desktop machine shows similar trends, albeit on a smaller scale. Again, S_{move} performs better than S_{local} on edge cases.

2 A Case Study: Building the Linux Kernel

We present a case study of the workload that led us to discover the frequency inversion phenomenon: building the Linux kernel version 5.4 with 320 jobs (`-j`) on a 4-socket Intel® Xeon E7-8870 v4 machine with 80 cores (160 hardware threads), with a nominal frequency of 2.1 GHz. Thanks to the Intel® SpeedStep and Turbo Boost technologies, our CPUs can individually vary the frequency of each core between 1.2 and 3.0 GHz. The frequency of the two hardware threads of a core is the same. In the rest of the paper, for simplicity, we use the term "core" for hardware threads.

Figure 1 shows the frequency of each core of the machine while the kernel build workload is running. This plot was produced with two tools that we have developed, SchedLog and SchedDisplay [10]. SchedLog collects the execution trace of an application with very low overhead. SchedDisplay produces a graphical view of such a trace. We have used SchedDisplay to generate all execution traces presented in this paper. SchedLog records the frequency information shown in

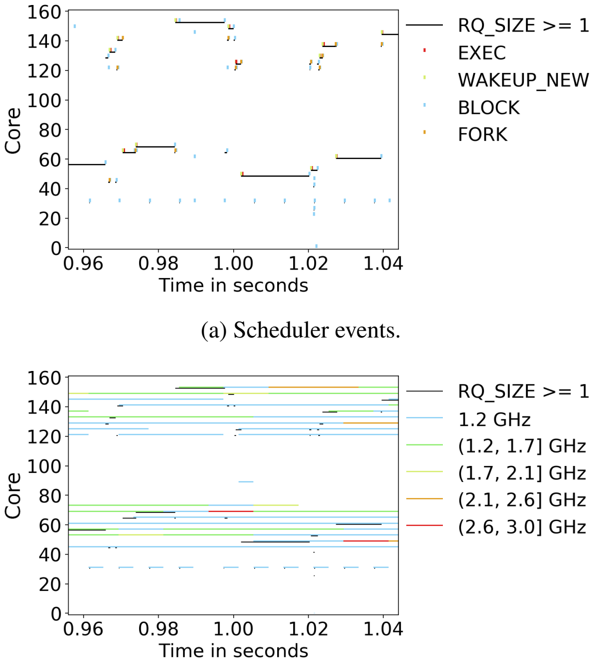
Figure 1 at each tick event (4ms in CFS). Consequently, the absence of a colored line in such traces means that ticks have been disabled by CFS on that core. CFS disables ticks on inactive cores to allow them to switch to a low-power state.

In Figure 1, we notice different phases in the execution. For a short period around 2 seconds, for a longer period between 4.5 and 18 seconds, and for a short period around 28 seconds, the kernel build has highly parallel phases that use all of the cores at a high frequency. The second of these three phases corresponds to the bulk of the compilation. In these three phases, the CPUs seem to be exploited to their maximum. Furthermore, between 22 and 31 seconds, there is a long phase of mostly sequential code with very few active cores, of which there is always one running at a high frequency. In this phase, the bottleneck is the CPU's single-core performance.

Between 0 and 4.5 seconds, and between 18 and 22 seconds however, there are phases during which all cores are used but they run at the CPU's lowest frequency (1.2 GHz). Upon closer inspection, these phases are actually mainly sequential: zooming in reveals that while all cores are used across the duration of the phase, only one or two cores are used at any given time. This raises two questions: why are so many cores used for a nearly sequential execution, and why are those cores running at such a low frequency.

We focus on the first couple of seconds where core utilization seems to be suboptimal. Zooming around 1 second, we first look at runqueue sizes and scheduling events, as illustrated in Figure 2a. We are in the presence of a pattern that is typical of mostly-sequential shell scripts: processes are created through the `fork()` and `exec()` system calls, and generally execute one after the other. These processes can easily be recognized on Figure 2a as they start with *WAKEUP_NEW* and *EXEC* scheduler events. After the process that runs on Core 56 blocks around the 0.96 s mark, three such short-lived processes execute one after the other on Cores 132, 140, and 65. After that, two longer-running ones run on Core 69 around the 0.98 s mark, and on Core 152 between the 0.98 s and 1.00 s mark. This pattern goes on for the entire duration of the execution shown in Figure 2a, with tasks created one after the other on Cores 148, 125, 49, 52, 129, 156, 60 and finally 145.

Looking at the core frequencies in the same part of the execution, as illustrated by Figure 2b, gives us a hint as to why cores are running slowly in this phase: there seems to be a significant delay between the time when a task starts running on a core, and the time when the core frequency starts increasing. For instance, between 1.00 s and 1.02 s, the task on Core 49 runs at a low frequency, and only when it is over at around 1.04 s does the frequency of the core rise to its maximum—before starting to decrease again almost instantly as the hardware notices that no task is running anymore on that core. The same issue can be observed shortly before 1.00 s on Core 152, and around 0.98 s on Core 69. In this last example, the core's frequency was even on a downward slope when the task started, and the frequency keeps going down even after the task ended before finally increasing again around 1.00 s. It appears that in the considered phase of the execution, the FTL is much higher than the duration of the tasks. Since tasks that follow each other tend to be scheduled on different cores, they are likely to always run at a low frequency as most cores are idle most of the time in this phase of the execution.



(a) Scheduler events.

(b) Core frequencies.

Figure 2: Zoom over a sparse region of Figure 1.

To confirm our intuition about the FTL, we develop a fine-grained tool [1] to monitor the frequency of a single core around the execution of an isolated busy loop, using the `powersave` governor. As shown in Figure 3, the task runs for 0.20 s, as illustrated by the *start* and *end* vertical lines in the figure. It takes an FTL of 29 ms for the core to go from its minimum frequency of 1.25 GHz to its maximum frequency of 3.00 GHz in order to accommodate the task. When the task ends, it takes approximately 10 ms for the core to go back to its initial frequency, but the duration of the FTL is compounded by the fact that the frequency tends to bounce back several times for around 98 ms before stabilizing at the core's lowest frequency. These measurements are consistent with our interpretation of Figure 2b: a FTL of several dozens of milliseconds is significantly longer than the execution of the tasks that are visible in the figure, as the longest task runs for around 20 ms between the 1.00 s and 1.02 s marks. Note that the duration of the FTL is mainly due to the time for the hardware to detect a load change and then decide to change the frequency. Previous work [22] shows that the actual latency for the core to change its frequency is only

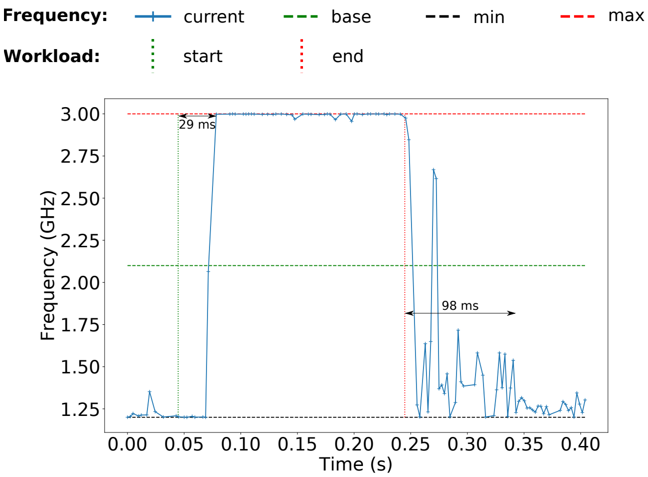


Figure 3: FTL for the Xeon E7-8870 v4 CPU.

tens of microseconds on Intel® CPUs.

Coming back to Figure 2a, the phenomenon we have been observing is the following. Computations in the (near) sequential phases of the build of Linux are launched sequentially as processes through the `fork()` and `wait()` system calls, and the execution of these computations is shorter than the FTL. Consequently, cores speed up after they have performed a computation, even though at that point, computation has moved to newly `fork`ed processes, which are likely to run on cores that were not recently used if the machine is not very busy. Indeed, CFS often selects different cores for tasks to wake up on, and if most cores are idle, it is likely that the selected cores were not used recently, and therefore run at a low frequency. The tasks that initiated the `fork()` perform `wait()` operations shortly afterwards, which means that the frequency increase they initiated is mostly wasted. We are in the presence of recurring *frequency inversion*, which is caused by a very common scenario: launching a series of sequential processes, as is commonly done in a shell script.

Sequential creation of processes through the `fork()` and `wait()` system calls is not the only cause of recurring frequency inversion. This phenomenon can also occur with lightweight threads that unblock and block each other, as is common in producer-consumer applications. Indeed, the CFS code that selects a core for a new task to wake up on is also used to select a core for already existing tasks that wake up. Note that CFS does not use different code paths depending on the type of task, namely, a process or a thread.

3 Strategies to Prevent Frequency Inversion

Since frequency inversion is the result of scheduling decisions, we believe it must be addressed at the scheduler level. In our experience, every change to the scheduler may have unpredictable consequences on some workloads, and the more complex the change, the less predictable the consequences. Therefore, proposing extensive or complex changes to the scheduler, or a complete rewrite, would make it unclear where performance gains come from. Striving for minimal, simple changes allows for an apples-to-apples comparison with CFS.

We propose two strategies to solve the frequency inversion problem. The first one is a simple strategy that offers good performance but suffers from large performance degradations in some scheduling scenarios. The second solution aims to have the same benefits as the first solution while minimizing worst cases at the expense of some simplicity.

3.1 Placing Threads Locally

The first strategy that we propose to prevent frequency inversion is S_{local}: when a thread is created or unblocked, it is placed on the same core as the process that created or unblocked it. In the context of creating a single process through the `fork()` and `wait()` system calls, this strategy implies that the created process is more likely to run on a high-frequency core, as the frequency of the core may already be high due to the activity of the parent. Furthermore, the duration in which there are two processes running on the same core will be limited, if the parent process calls `wait()` shortly afterwards. In the context of a producer-consumer application, when a producer thread wakes up a consumer thread, this strategy again implies that the consumer thread is more likely to run on a high-frequency core, and the duration in which there are two processes running on the same core will again be limited, if the last action of the producer is to wake up the consumer before blocking or terminating.

However, there are cases in which S_{local} might hurt performance: if the task that created or woke another task does not block or terminate quickly afterwards, the created or woken task will wait for the CPU resource for a certain period of time. This issue is mitigated by the periodic load balancer of the Linux scheduler that will migrate one of the tasks to another less loaded core. However, waiting for the next load balancing event might be quite long. In CFS, periodic load balancing is performed hierarchically, with different periods: cores in the same cache domain are more frequently balanced than cores on different NUMA nodes. These periods can vary from 4 to hundreds of milliseconds on large machines.

S_{local} significantly changes the behavior of CFS by fully replacing its thread placement strategy. Additionally, the aforementioned shortcomings make it a high risk solution for certain workloads. Both issues make this solution unsatisfactory given the prerequisites that we previously set.

3.2 Deferring Thread Migrations

In order to fix core oversubscription without waiting for periodic load balancing, we propose a second strategy, S_{move}. With vanilla CFS, when a thread is created or woken, CFS

CPU vendor	Intel®	AMD®
CPU model	Xeon E7-8870 v4	Ryzen 5 3400G
Cores (SMT)	80 (160)	4 (8)
Min freq	1.2 GHz	1.4 GHz
Base freq	2.1 GHz	3.7 GHz
Turbo freq	3.0 GHz	4.2 GHz
Memory	512 GB	8 GB
OS	Debian 10 (buster)	Arch Linux

Table 1: Configurations of our experimental machines.

decides on which core it should run. S_{move} defers the use of this chosen core to allow waking threads to take advantage of a core that is more likely to run at a high frequency.

Let T_{woken} be the newly created or waking task, C_{waker} the core where task T_{waker} that created or woke T_{woken} is running and C_{CFS} the destination core chosen by CFS. The normal behavior of the scheduler is to directly enqueue task T_{woken} into C_{CFS}'s runqueue. We propose to delay this migration to allow T_{woken} to be more likely to use a high-frequency core if C_{CFS} is running at a low frequency. First, if C_{CFS} is running at a frequency higher than the CPU's minimum one, we enqueue T_{woken} in C_{CFS}'s runqueue. Otherwise, we arm a high-resolution timer interrupt that will perform the migration in D μs and we enqueue T_{woken} into C_{waker}'s runqueue.The timer is cancelled if T_{woken} is scheduled on C_{waker}.

The rationale behind S_{move} is that we want to avoid waking low frequency cores if the task can be performed quickly when placed locally on a core that is likely to run at a high frequency. Indeed, T_{waker} is running at the time of the placement, meaning that C_{waker} is likely to run at a high frequency. The delay D can be changed at run time by writing to a parameter file in the `sysfs` pseudo file system. We have chosen a default value of 50 μs, which is close to the delay between a fork and a wait system call during our Linux kernel build experiments. We have found that varying the value of this parameter between 25 μs and 1 ms has insignificant impact on the benchmarks used in Section 4.

4 Evaluation

This section aims to demonstrate that our strategies improve performance on most workloads, while not degrading energy consumption. We run a wide range of applications from the Phoronix benchmark suite [23], the NAS benchmark suite [5], as well as other applications, such as `hackbench` (a popular benchmark in the Linux kernel scheduler community) and `sysbench OLTP` (a database benchmark). These experiments are run on a server-grade 4-socket NUMA machine with an 80-core Intel® CPU and on a desktop machine equipped with a 4-core AMD® CPU (Table 1). Both CPUs can select independent frequencies for each core[1] We have implemented S_{local} and S_{move} in the latest LTS kernel, Linux 5.4, released in November 2019 [3], and compare our strategies to Linux 5.4.

Implementing S_{local} (resp. S_{move}) only required modifying 3 (resp. 124) lines in CFS. We run all experiments 10 times. Energy consumption is evaluated on both machines using the Intel® RAPL [19] feature, which measures the energy consumption of the CPU socket and the DRAM. The performance results are those reported by each benchmark, and thus they involve different metrics, such as execution time, throughput, or latency, with inconsistent units. For better readability, all the following graphs show the improvement in terms of performance and energy usage compared to the mean of the runs with CFS. Therefore, higher is always better, regardless of the measured unit. The mean of the results for CFS is displayed on top of the graph for all benchmarks with the benchmark's unit.

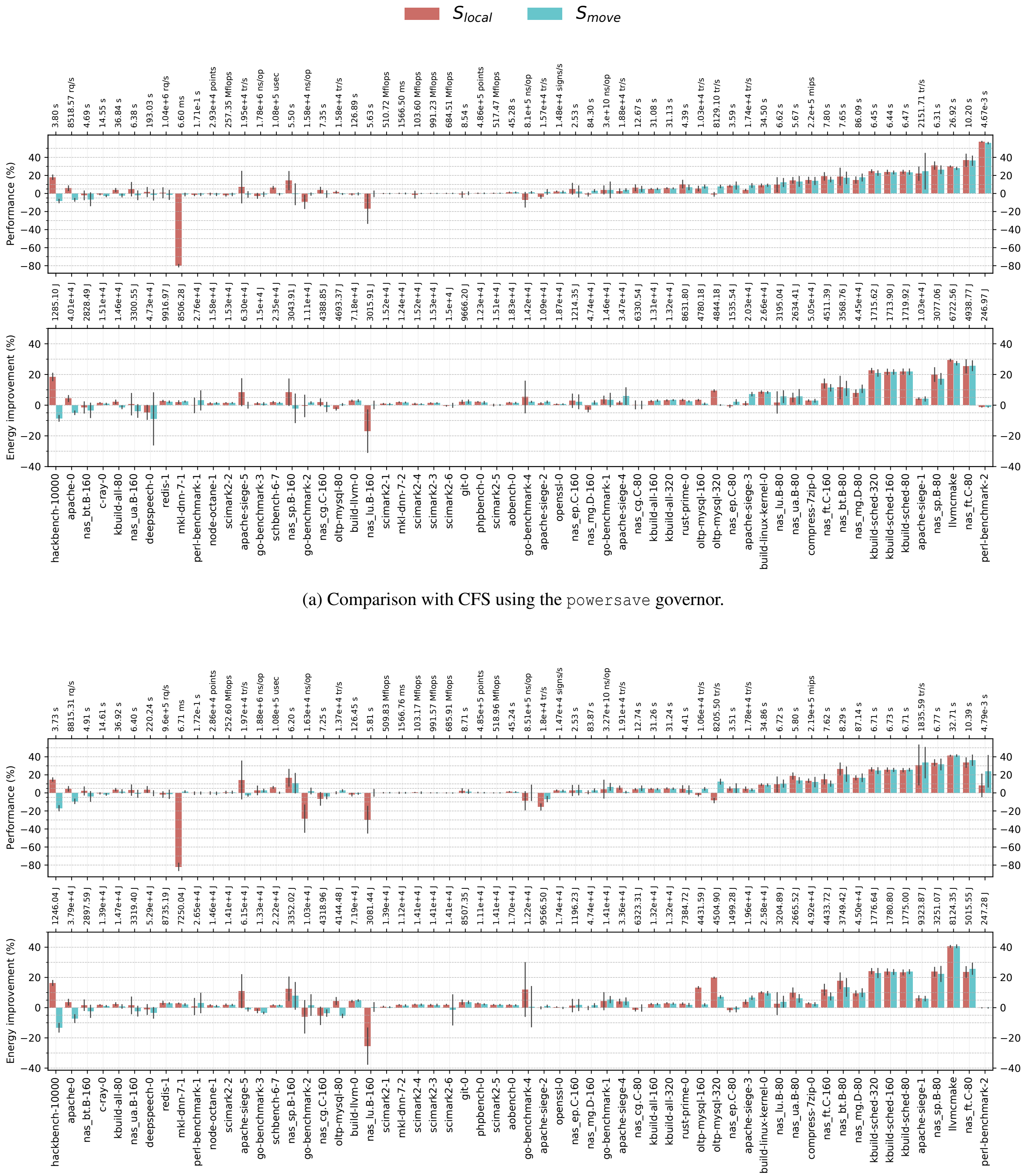
In Linux, frequency is controlled by a subsystem called a *governor*. On modern Intel® hardware, the `powersave` governor delegates the choice of the frequency to the hardware since it can perform more fine-grained adjustments. The hardware frequency-selection algorithm tries to save energy with a minimal impact on performance. The hardware estimates the load of a core based on various heuristics such as the number of retired instructions. This is the default governor for Intel® hardware on most Linux distributions. The `schedutil` governor, in development by the Linux community since Linux 4.7 (July 2016), tries to give control back to the operating system. It uses the internal data of the kernel scheduler, CFS, to estimate the load on each core, and changes the frequency accordingly. Two other governors, `performance` and `ondemand`, are available in Linux but are of no interest to us: the former runs all cores at the highest frequency, thus disabling dynamic scaling, while the latter is not supported on modern Intel® processors. To demonstrate that our work is orthogonal to the used governor, we evaluate our strategies using both `powersave` and the `schedutil`.

We first present the complete results on the Intel® server and summarize the results on the AMD® desktop machine. We then revisit our kernel build case study and study some worst case results (`mkl`, `hackbench`). Finally, we discuss the overhead of our S_{move} strategy.

4.1 Execution Using `powersave`

We first consider the execution under `powersave`. Figure 4a shows the improvement in terms of performance and energy consumption of S_{local} and S_{move} as compared to CFS. We consider that improvements or deteriorations that do not exceed 5% to be on par with CFS.

[1] This is different from turbo frequencies: many desktop and laptop CPUs have per-core DFS in order to support turbo frequencies, but in practice, all cores not using the turbo range run at the same frequency.



(a) Comparison with CFS using the `powersave` governor.

(b) Comparison with CFS using the `schedutil` governor.

Figure 4: Performance and energy consumption improvement w.r.t. Linux 5.4 on the server machine (higher is better).

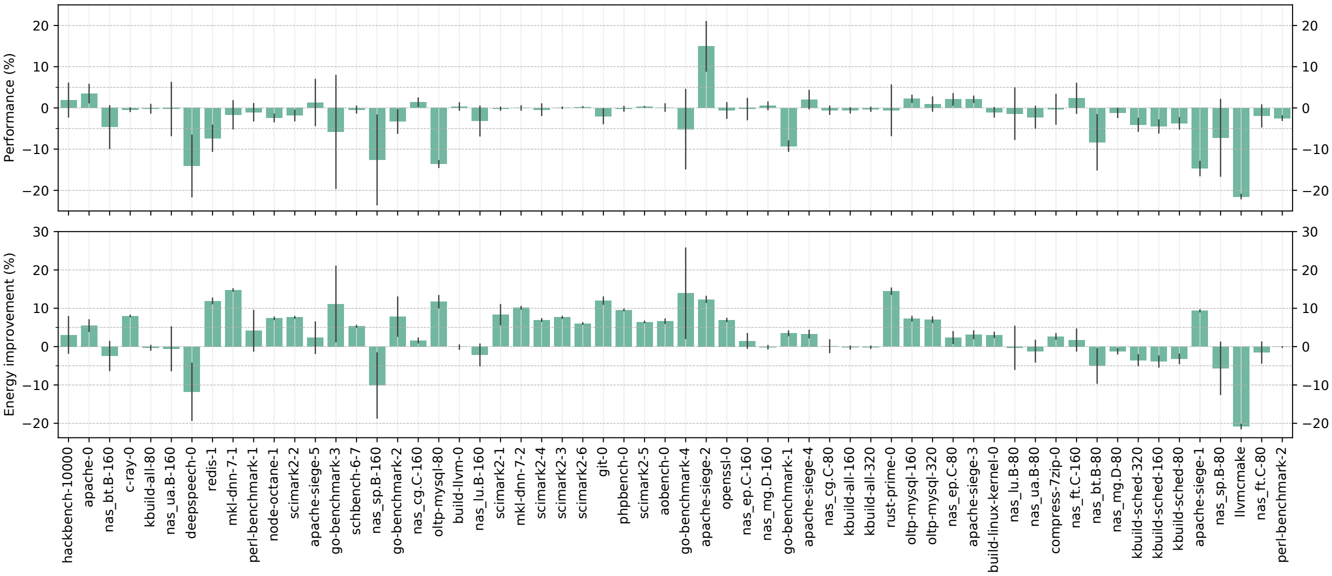


Figure 5: Performance of `schedutil` compared to `powersave` with CFS on the server machine.

Performance. Both S_{local} and S_{move} perform well overall with respectively 27 and 23 out of 60 applications outperforming CFS. The best results for these policies are seen, as expected, on benchmarks that extensively use the `fork/wait` pattern, and therefore exhibit a large number of frequency inversions. In the best case, S_{local} and S_{move} gain up to 58% and 56% respectively on `perl-benchmark-2`, that measures the startup time of the `perl` interpreter. This benchmark benefits greatly from avoiding frequency inversions since it mostly consists of `fork/wait` patterns. In terms of performance losses, both strategies deteriorate the performance of only 3 applications, but on very different scales. S_{local} deteriorates `mkl-dnn-7-1` by 80% and `nas_lu.B-160` by 17% while S_{move} has a worst case deterioration of 8.4% on `hackbench`.

Energy consumption. Overall, both S_{local} and S_{move} improve energy usage. Out of our 60 applications, we improve energy consumption by more than 5% for 16 and 14 applications, respectively, compared to CFS. Most of the improvements are seen on benchmarks where performance is also improved. In these cases, the energy savings are likely mostly due to the shorter execution times of the applications. However, we also see some improvements on applications where the performance is on par with that on CFS. This is due to the fact that we avoid waking up cores that are in low power states, therefore saving the energy necessary to power up and run those cores. In terms of loss, S_{local} consumes more energy than CFS on only one application, `nas_lu.B-160`. This loss is explained by the bad performance of S_{local} on this application. This benchmark's metric is its execution time, and increasing the execution time without correspondingly reducing the frequency increases the energy consumption. S_{move} consumes more energy than CFS on two applications: `hackbench`, because of the performance loss, and `deepspeech` that has too high a standard deviation for its results to have significance.

Overall score. To compare the overall impact of our strategies, we compute the geometric mean of all runs, where each run is normalized to the mean result of CFS. S_{move} has a performance improvement of 6%, a reduction in energy usage of 3% and an improvement of 4% with both metrics combined. S_{local} has similar overall scores (always 5%), but its worst cases suggest that S_{move} is a better option for a general-purpose scheduler. These small differences are expected because most of the applications we evaluate perform similarly with CFS and with our strategies. We also evaluate the statistical significance of our results with a t-test. With p-values of at most $3 \cdot 10^{-20}$, we deem our results statistically significant.

4.2 Execution Using `schedutil`

Next, we consider execution under the `schedutil` governor. As a baseline, Figure 5 first shows the performance and energy improvements of the `schedutil` governor compared to the `powersave` governor with CFS. Overall, we observe that the `schedutil` governor deteriorates the performance of most applications while improving energy usage. This indicates that this new governor is more aggressive in terms of power savings than the one implemented in hardware. We omit raw values since they are already available in Figures 4a and 4b. Figure 4b then shows the improvement in terms of performance and energy consumption of our strategies compared to CFS, when using the `schedutil` governor.

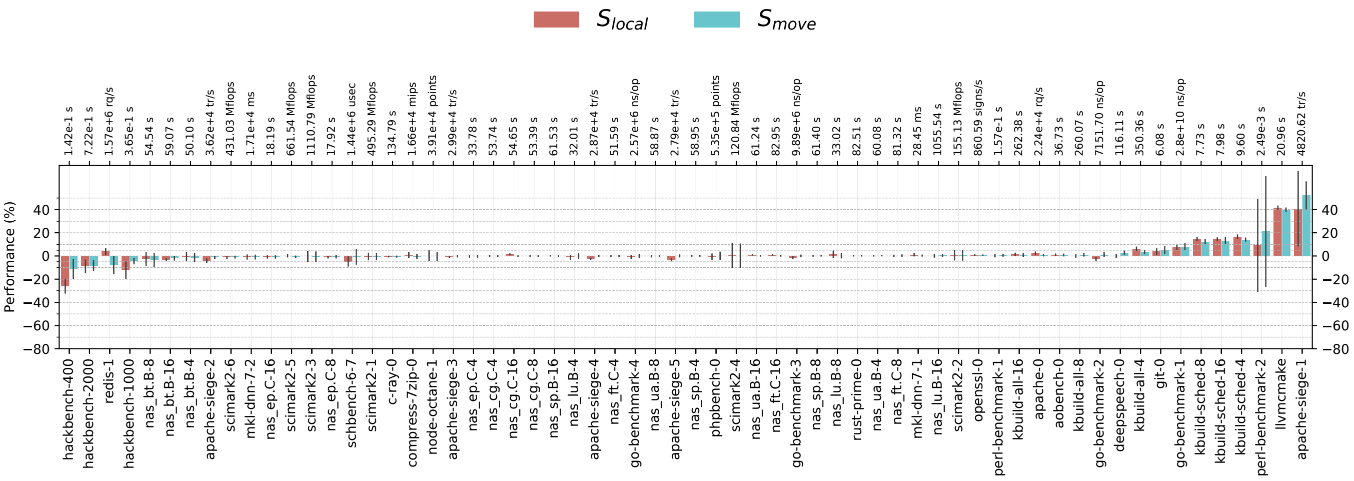


Figure 6: Performance improvement w.r.t. Linux 5.4 on the desktop machine (higher is better).

Performance. S_{local} and S_{move} outperform CFS on 22 and 20 applications out of 60 respectively. The applications concerned are the same that were improved with the `powersave` governor. In terms of performance losses, however, S_{local} is more impacted by the `schedutil` governor than S_{move}, with 7 applications performing worse than CFS versus only 2.

Energy consumption. The overall improvement in terms of energy usage of `schedutil` with CFS would suggest that we might see the same trend with S_{local} and S_{move}. And indeed, the results are quite similar to what we observe with the `powersave` governor.

Overall score. The geometric means with this governor are the following for `schedutil` and S_{move}: 6% for performance, 4% for energy and 5% with both metrics combined. S_{local} has similar results (2%, 6% and 4% respectively), but the worst cases are still too detrimental for a general-purpose scheduler. These results are also statistically significant with p-values of at most $3 \cdot 10^{-20}$.

4.3 Evaluation on the Desktop Machine

We also evaluate our strategies on the smaller 4-core AMD® desktop CPU presented in Table 1. In contrast to Intel® CPUs, the `powersave` governor on AMD® CPUs always uses the lowest available frequency, making it unusable in our context. We therefore use the `schedutil` governor on this machine.

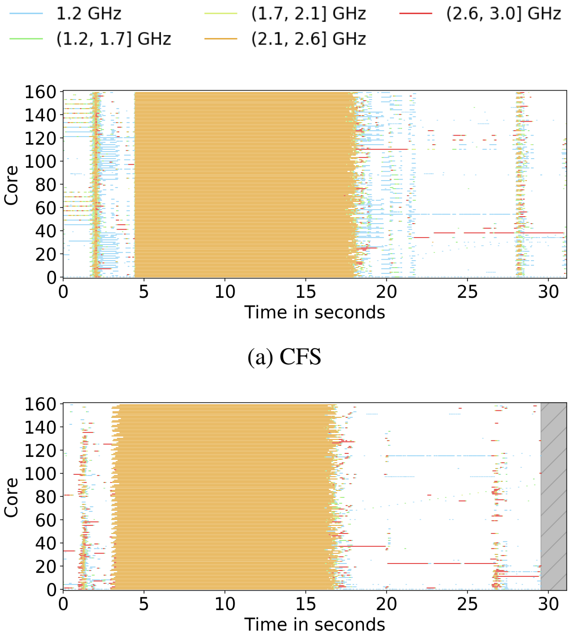
As shown in Figure 6, we observe the same general trend as on our server machine. S_{local} and S_{move} behave similarly when there is improvement, and S_{move} behaves better on the few benchmarks with performance degradation. We measure at worst an 11% slowdown and at best a 52% speedup for S_{move}, with an aggregate performance improvement of 2%. Additionally, S_{move} improves the performance of 7 applications by more than 5% while only degrading the performance of 4 applications at the same scale. The S_{local} strategy gives the same results regarding the number of improved and degraded applications, but suffers worse edge cases. Its best performance improvement is 42% while its worst deterioration is 25%, with an aggregate performance improvement of 1%. We conclude that even if there is no major global improvement, S_{move} is still a good strategy to eliminate frequency inversions on machines with smaller core counts. Our performance results are statistically significant, with p-values of $5 \cdot 10^{-4}$ for S_{move} and $3 \cdot 10^{-2}$ for S_{local}.

In terms of energy consumption, both S_{local} and S_{move} seem to have little to no impact as compared to CFS. However, the measures we were able to gather with all three strategies had a large variance that we did not observe on our Intel® CPU. We suspect that this is due to the energy-related hardware counters available on AMD® processors or the lack of good software support for these counters.

4.4 In-Depth Analysis

We now present a detailed analysis of specific benchmarks that either performed particularly well or particularly poorly with our solutions. In this section all traces were obtained with the `powersave` governor.

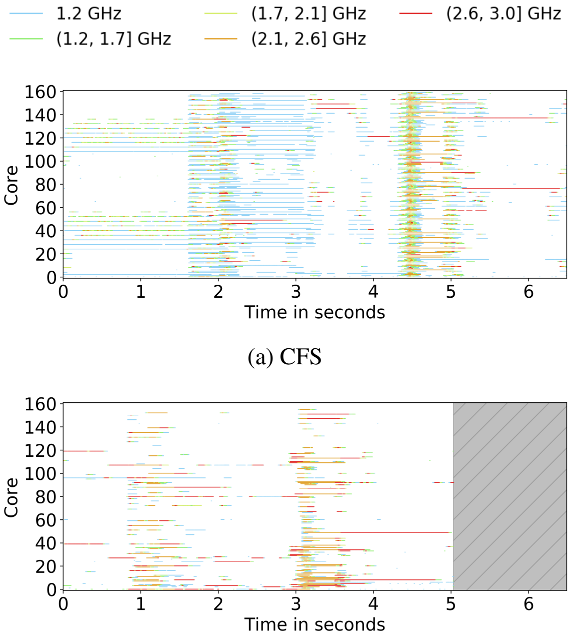
`kbuild` Figure 7 shows the execution of the build of the Linux kernel as presented in the case study, with CFS (top) and S_{move} (bottom). During the mostly sequential phases with multiple cores running at a low frequency on CFS (0-2 s, 2.5-4.5 s, 17-22 s), S_{move} uses fewer cores at a higher frequency. This is mainly due to the `fork()`/`wait()` pattern: as the waker thread calls `wait()` shortly after the `fork()`, the S_{move} timer does not expire and the woken threads remain on the local core running at a high frequency, thus avoiding fre-



(a) CFS

(b) S_{move}

Figure 7: Execution trace when building the Linux kernel version 5.4 using 320 jobs.



(a) CFS

(b) S_{move}

Figure 8: Execution trace when building the `sched` directory of the Linux kernel version 5.4 using 320 jobs.

quency inversion. As a result, for example, the phase before the long parallel phase is executed in 4.4 seconds on CFS and in only 2.9 seconds with S_{move}.

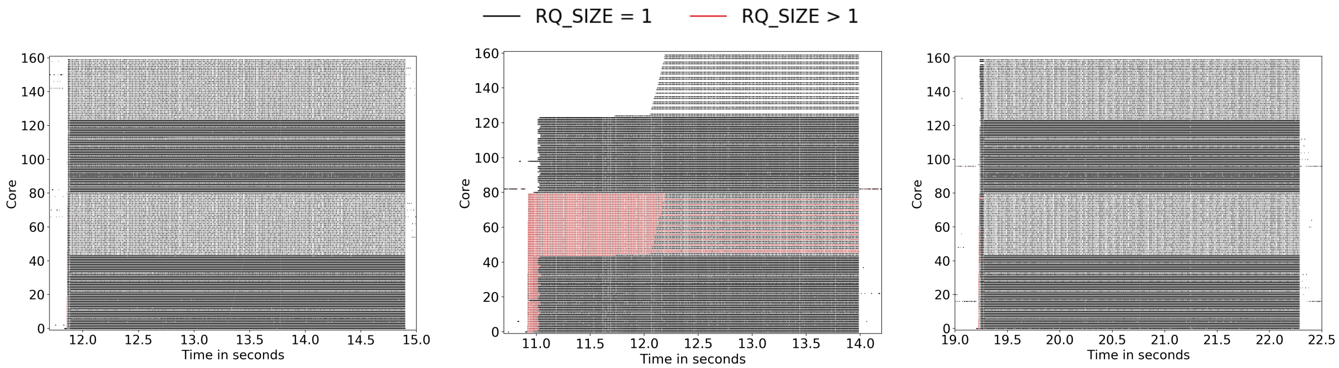
To understand the impact of S_{move} better, Figure 8 shows the `kbuild-sched-320` benchmark, which builds only the scheduler subsystem of the Linux kernel. Here, the parallel phase is much shorter than with a complete build, as there are fewer files to compile, making the sequential phases of the execution more visible. Again, we see that fewer cores are used, at a higher frequency.

`mkl` The `mkl-dnn-7-1` benchmark is the worst-case scenario for S_{local}: all threads keep blocking and unblocking and therefore avoid periodic load balancing and continue returning to the same set of cores. Thus, threads that are sharing a core with another thread will tend to remain there with the S_{local} strategy. Figure 9 shows the number of threads on the runqueue of each core with all three schedulers with the `powersave` governor. A black line indicates that there is one thread in the runqueue, and a red line indicates that there is more than one. CFS spreads the threads on all cores rapidly, and achieves a balanced machine with one thread per core in less than 0.2 seconds. On the other hand, S_{local} tries to maximize core reuse and oversubscribes 36 cores. This leads to never using all cores, achieving at most 85% CPU utilization with multiple cores overloaded. This is a persistent violation of the *work-conservation* property, as defined by Lozi *et al.* [21], *i.e.*, no core is idle if a core has more than one thread in its runqueue.

Interestingly, in our experiment, the balancing operations that spread threads are due to system or daemon threads (e.g. `systemd`) that wake up and block immediately, thus triggering an idle balancing from the scheduler. On a machine with nothing running in the background, we could have stayed in an overloaded situation for a long period of time, as ticks are deactivated on idle cores, removing opportunities for periodic balancing. We can see the same pattern on `nas-lu.B-160`, another benchmark that does not work well with S_{local}. S_{move} solves the problem by migrating, after a configurable delay, the threads that overload cores to available idle cores.

`hackbench` The `hackbench-10000` benchmark is the worst application performance-wise for the S_{move} strategy. This micro-benchmark is particularly stressful for the scheduler, with 10,000 running threads. However, the patterns exhibited are interesting to better understand the shortcomings of S_{move} and give insights on how to improve our strategies.

This benchmark has three phases: thread creation, communication and thread termination. Figure 10 shows the frequency of all cores during the execution of `hackbench` with CFS, S_{local} and S_{move}. The first phase corresponds to the first two seconds on all three schedulers. A main thread creates 10,000 threads with the `fork()` system call, and all child threads immediately wait on a barrier. With CFS, child threads are placed on idle cores that become idle again when the threads arrive at the barrier. This means that all cores remain

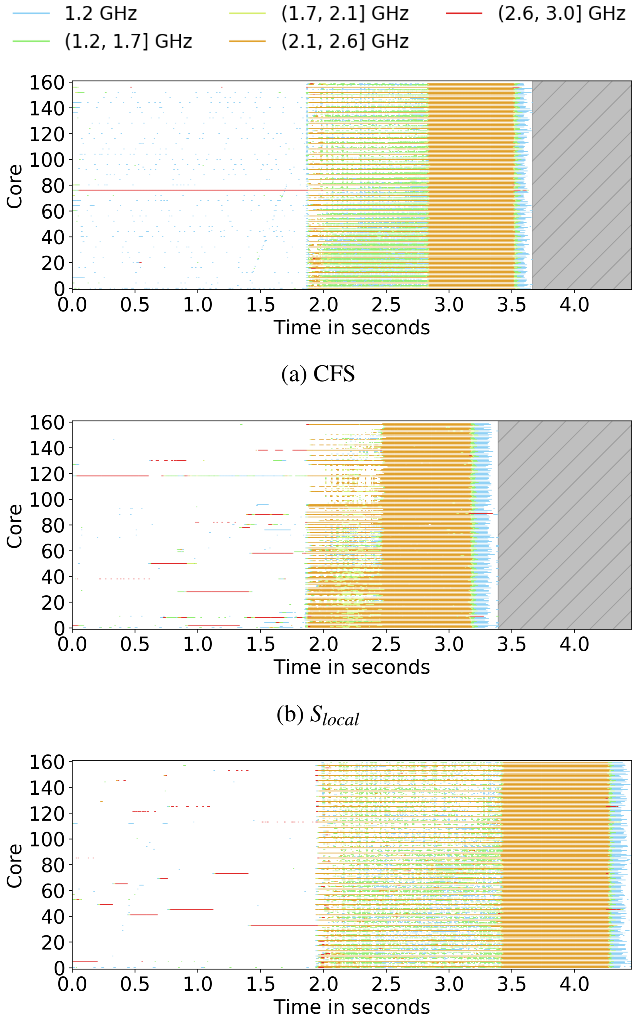


(a) CFS

(b) S_{local}

(c) S_{move}

Figure 9: Number of threads per core during the execution of `mkl-dnn-7-1`.



(a) CFS

(b) S_{local}

(c) S_{move}

Figure 10: Core frequency when executing `hackbench`.

mostly idle. This also leads to the main thread remaining on the same core during this phase. However, S_{local} and S_{move} place the child threads locally, causing oversubscription of the main thread's core and migrations by the load balancer. The main thread itself is thus sometimes migrated from core to core. When all threads are created, the main thread releases the threads waiting on the barrier and waits for their termination, thus beginning the second phase. During this phase, the child threads communicate by reading and writing in pipes. CFS tries to even out the load between all cores, but its heuristics give a huge penalty to migrations across NUMA nodes, so a single node runs at a high frequency (cores 0, 4, 8, etc. share the same node on our machine) while the others have little work to perform and run at lower frequencies. This phase finishes at 2.8 seconds. The remainder of the execution is the main thread reaping its children and terminating.

S_{local} packs threads aggressively, leading to long runqueues in the second phase, and therefore facilitating load balancing across nodes because of the large induced overload. However, S_{local} still does not use all cores, mainly avoiding running on hyperthreaded pairs of cores (cores n and $n+80$ are hyperthreaded on our machine). S_{local} runs the second phase faster than CFS, terminating it at 2.5 seconds, because it uses half of the cores at a high frequency all the time, and many of the other cores run at a medium frequency.

On the other hand, S_{move} performs poorly in the second phase, completing it at 3.4 seconds. The behavior seems very close to that of CFS, with one core out of four running at a high frequency. However, S_{move} results in more idleness or low frequency on the other cores. This is due to S_{move} placing threads locally: many threads contend for the local core; some are able to use the resource while others are migrated when the timer interrupt is triggered. The delays cause idleness compared to CFS, and the migrations leave cores idle, lowering their frequency compared to S_{local}. Additionally, when threads are migrated because of timers expiring, they are all placed on the same core, and oversubscribe it. For `hackbench`, choosing the middle ground is the worst strategy. We can also note that load balancing is not able to mitigate this situation because of the high volatility of this workload. This problem was also demonstrated by Lozi et al. [21] on a database application.

This hackbench setup is an extreme situation that is unlikely to happen in real life, with a largely overloaded machine (10,000 threads) and a highly volatile application. This microbenchmark is only interesting to study the behavior of our strategies. Still, overall, S_{move} gives better performance than S_{local}.

4.5 Scheduling Overhead of S_{move}

S_{move} is more complex than S_{local}, and so we analyze its overhead as compared to CFS, as an upper bound for our strategies. We identify two possible sources of overhead: querying frequency and using timers.

First, we evaluate the cost of querying the core frequency. Querying the frequency of a core mostly consists in reading two hardware registers and performing some arithmetic operations, as the current frequency is the division of these two registers times the base frequency of the CPU. Even though this is a very small amount of computation compared to the rest of the scheduler, we minimize it furthermore by querying this information at every tick instead of every time it is needed. In our benchmarks, we notice no difference in performance with or without querying the frequency at every tick.

Second, we evaluate the cost of triggering a large number of timers in the scheduler. To do so, we run schbench on two versions of Linux: the vanilla 5.4 kernel and a modified version with timers armed under the same condition as S_{move}. Here, however, the timer handler does not migrate the thread as in S_{move}. We choose schbench because it performs the same workload as hackbench but provides, as a performance evaluation, the latencies of the messages sent through pipes instead of the completion time. Table 2 shows the results of this benchmark. Overall, the 99.5^{th} percentile of latencies is the same for both versions of the kernel, except for 256 threads where timers have a negative impact. We can also observe that the number of timers triggered increases with the number of threads but drops after 256 threads. This behavior is expected: more threads means more wake-ups, but when the machine starts being overloaded, all cores run at high frequencies, and the timers are less frequently armed. This tipping point arrives around 256 threads because schbench threads constantly block, meaning that typically fewer than 160 threads are runnable at a time.

5 Discussion

As previously stated, our proposed solutions S_{local} and S_{move} are purposefully simple. We now discuss other more complex solutions to the frequency inversion problem.

Threads	Latency		Timers triggered
	vanilla	with timers	
64	78	77	2971
128	86	84	13910
192	119	144	63965
256	2292	3188	93001
512	36544	36544	512
768	60224	60480	959
1024	76416	76928	1290

Table 2: schbench latencies (99.5^{th} percentile, in μsec) and number of timers triggered.

High frequency pool. A possible solution would be to keep a pool of cores running at a high frequency even though no thread is running on them. This would allow threads to be placed on an idle core running at a high frequency instantaneously. This pool could, however, waste energy and reduce the maximal frequency attainable by busy cores, which diminishes when the number of active cores increases.

Tweaking the placement heuristic. We could add a new frequency heuristic to the existing placement strategy. However, the tradeoff between using a core running at a higher frequency and e.g., cache locality is not clear, and may vary greatly according to the workload and the architecture.

Frequency model. The impact of the frequency of one core on the performance of other cores is hardware-specific. If the scheduler were to take frequency-related decisions, it would also need to account for the impact its decision would have on the frequency of all cores. Such models are not currently available, and would be complicated to create.

6 Related Work

Dynamic frequency scaling. Using DFS to reduce energy usage has been studied for over two decades. Weiser *et al.* [33] were the first to propose to adjust the frequency of the CPU according to its load, with the aim to maximize the *millions of instructions per joule* metric. Following this, in the early 2000s, Chase *et al.* [11] as well as Elnozahy *et al.* [17] proposed to reduce the frequency of underutilized servers in farms that exhibit workload concentration. Bianchini and Rajamony summarized these early works in a survey from 2004 [6]. Nowadays, on the hardware side, most CPUs support DFS, with the most recent series having elaborate hardware algorithms that are able to dynamically select very different frequencies for cores on the same chip, with technologies such as Enhanced Intel SpeedStep® [2] and AMD® SenseMI [4]. Despite this shift of DFS logic from the software side to the hardware side in recent years, the decision to develop the experimental `schedutil` [9] governor in Linux was based on the idea that software still has a role to play in DFS, as it knows better the load being executed. Similarly, our strategies show that the software placing tasks on high-frequency cores can be more efficient than waiting for the hardware to increase the frequency of cores after task placement, due to the FTL.

Tracking inefficient scheduler behavior. Perf [15, 16, 32], which is provided with the Linux kernel, supports monitoring scheduler behavior through the `perf sched` command. While `perf sched` makes it possible to analyze the behavior of the scheduler on simple workloads with good accuracy, it has significant overhead on the Linux kernel build and other real-world workloads. Lozi et al. [21] identify performance bugs in the Linux scheduler. To analyze them, they write a basic profiler that monitors, for each core, the number of queued threads and the load. Their basic profiler does not monitor scheduling events. SchedLog and SchedDisplay [10], which we use in this paper, make it possible to record relevant information about all scheduler events with low overhead, and to efficiently navigate through the large amount of recorded data with a powerful and scriptable graphical user interface.

Mollison et al [25] apply regression testing to schedulers. Their focus is limited to real-time schedulers, and they do not take DFS into account. More generally, there has been an ongoing effort to test and understand the impact of the Linux scheduler on performance. Since 2005, the LKP project [12] has focused on hunting performance regressions, and a myriad of tools that make it possible to identify performance bugs in kernels have been proposed by the community [7, 18, 26, 28]. The focus of these tools, however, is to detect slowdowns inside the kernel code, and not slowdowns in application code that were caused by decisions from the kernel. Consequently, they are unable to detect poor scheduling behavior.

Improving scheduler behavior. Most previous work focuses on improving general-purpose OS scheduling with new policies that improve a specific performance metric, such as reducing contention over shared resources [31, 35], optimizing the use of CPU caches [29, 30], improving NUMA locality [8, 14] or minimizing idleness [20]. These papers systematically disable DFS in their experiments. Merkel *et al.* [24] propose a scheduling algorithm that avoids resource contention by co-scheduling applications that use complementary resources. They reduce contention by lowering the frequency of cores that execute inauspicious workloads. Zhang et al. [34] propose a scheduling policy for multi-core architectures that facilitates DFS, although their main focus is reducing cache interference. They only consider per-chip DFS, as per-core DFS was not commonplace at the time.

Linux kernel developers have recently focused on DFS and turbo frequencies [13], as it was discovered that a short-lived jitter process that runs on a previously idle core can make that core switch to turbo frequencies, which can in turn reduce the frequencies used by other cores—even after the jitter process completes. To solve this issue, a patch [27] was proposed to explicitly mark jitter tasks. The scheduler then tries to place these marked tasks on cores that are active and expected to remain active. In contrast, the frequency inversion issue we identified is not specifically caused by turbo frequencies: it can occur with any DFS policy in which different cores may run at different frequencies.

Child runs first. CFS has a feature that may seem related to our solutions: `sched_child_runs_first`. At thread creation, this feature assigns a lower *vruntime* to the child thread, giving it a higher priority than its parent. If CFS places the thread on the same core as its parent, the thread will preempt the parent; otherwise, the thread will just run elsewhere. This feature does not affect thread placement and thus cannot address the frequency inversion problem. Using this feature in combination with S_{move} would defeat S_{move}'s purpose by always canceling the timer. The strategy would resemble S_{local}, except that the child thread would always preempt its parent.

7 Conclusion

In this paper, we have identified the issue of frequency inversion in Linux, which occurs on multi-core CPUs with per-core DFS. Frequency inversion leads to running tasks on low-frequency cores and may severely degrade performance. We have implemented two strategies to prevent the issue in the Linux 5.4 CFS scheduler. Implementing these strategies required few code changes: they can easily be ported to other versions of the Linux kernel. On a diverse set of 60 applications, we show that our better solution, S_{move}, often significantly improves performance. Additionally, for applications that do not exhibit the frequency inversion problem, S_{move} induces a penalty of 8% or less with 3 of the evaluated applications. As independent core frequency scaling becomes a standard feature on latest generation processors, our work will target a larger number of machines.

In future work, we want to improve thread placement in the scheduler by including the cores' frequencies directly in the placement algorithm. This improvement will need to account for various parameters such as architecture-specific DFS, simultaneous multi-threading and maintaining cache locality.

Acknowledgments and Availability

This work is supported in part by Oracle donation CR 1930. We would also like to thank the anonymous reviewers and our shepherd, Heiner Litz, for their feedback.

S_{local} and S_{move} patches for Linux 5.4 are available at: https://gitlab.inria.fr/whisper-public/atc20.

References

[1] frequency_logger. https://github.com/rgouicem/frequency_logger.

[2] Intel®. Frequently Asked Questions about Enhanced Intel SpeedStep® Technology for Intel® Processors. https://www.intel.com/content/www/us/en/support/articles/000007073/processors.html.

[3] Linus Torvalds' official git repository. https://github.com/torvalds/linux.

[4] AMD®. SenseMI Technology. https://www.amd.com/en/technologies/sense-mi.

[5] D.H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R.A Fatoohi, P. O. Frederickson, T. A Lasinski, R. S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing*, pages 158–165, Seattle, WA, USA, 1991.

[6] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–76, 2004.

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *OSDI*, pages 86–93, Vancouver, BC, Canada, 2010.

[8] Timothy Brecht. On the importance of parallel application placement in NUMA Multiprocessors. In *USENIX SEDMS*, San Diego, CA, USA, 1993.

[9] Neil Brown. Improvements in CPU frequency management. https://lwn.net/Articles/682391/.

[10] Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. Fork/wait and multicore frequency scaling: a generational clash. In *PLOS*, pages 53–59, Huntsville, ON, Canada, 2019. ACM.

[11] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS operating systems review*, 35(5):103–116, 2001.

[12] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, pages 93–102, Ottawa, ON, Canada, 2007.

[13] Jonathan Corbet. TurboSched: the return of small-task packing. *Linux Weekly News*, July 1, 2019. https://lwn.net/Articles/792471/.

[14] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *ASPLOS*, pages 381–394, Houston, TX, USA, 2013.

[15] Arnaldo Carvalho de Melo. Performance counters on Linux. In *Linux Plumbers Conference*, Portland, OR, USA, 2009.

[16] Arnaldo Carvalho de Melo. The new Linux 'perf' tools. In *Slides from Linux Kongress*, Nuremberg, Germany, 2010.

[17] EN Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *PACS*, pages 179–197, Cambridge, MA, USA, 2002. Springer.

[18] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Our troubles with Linux and why you should care. In *APSys*, pages 1–5, Shanghai, China, 2011.

[19] Intel®. Intel® and 64 and IA-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part 2, Chapter 14.9*, page 5, 2011.

[20] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema: application to work conservation. In *EuroSys*, pages 3:1–3:16, Heraklion, Greece, 2020. ACM.

[21] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: a decade of wasted cores. In *EuroSys*, pages 1–16, London, UK, 2016.

[22] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of CPU frequency transition latency. *Comput. Sci. Res. Dev.*, 29(3-4):187–195, 2014.

[23] Phoronix Media. Phoronix test suite – Linux testing & benchmarking platform, automated testing, open-source benchmarking. http://www.phoronix-test-suite.com/.

[24] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys*, pages 153–166, Paris, France, 2010. ACM.

[25] Malcolm S Mollison, Björn Brandenburg, and James H Anderson. Towards unit testing real-time schedulers in LITMUS^{RT}. In *OSPERT*, Stuttgart, Germany, 2009.

[26] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *SOSP*, pages 134–145, Asheville, NC, USA, 1993.

[27] Parth Shah. TurboSched: A scheduler for sustaining turbo frequencies for longer durations, June 25, 2019. https://lkml.org/lkml/2019/6/25/25.

[28] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O system performance debugging using model-driven anomaly characterization. In *FAST*, pages 309–322, San Francisco, CA, USA, 2005.

[29] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys*, pages 47–58, Lisbon, Portugal, 2007.

[30] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *HPCA*, pages 188–197, Shenzhen, China, 2013.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, pages 1–17, Bordeaux, France, 2015.

[32] Vincent M Weaver. Linux perf_event features and overhead. In *FastPath*, pages 80–87, Austin, TX, 2013.

[33] Mark Weiser, Brent B. Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *OSDI*, pages 13–23, Monterey, CA, USA, 1994.

[34] Xiao Zhang, Sandhya Dwarkadas, and Rongrong Zhong. An evaluation of per-chip nonuniform frequency scaling on multicores. In *USENIX ATC*, Berkeley, CA, USA, 2010.

[35] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.

vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds

Weiwei Jia[1]*, Jianchen Shan*[2]*, Tsz On Li*[3]*, Xiaowei Shang*[1] *Heming Cui*[3]*, Xiaoning Ding*[1]
[1]*New Jersey Institute of Technology* [2]*Hofstra University* [3]*University of Hong Kong*

Abstract

The paper focuses on an under-studied yet fundamental issue on Simultaneous Multi-Threading (SMT) processors — how to schedule I/O workloads, so as to improve I/O performance and efficiency. The paper shows that existing techniques used by CPU schedulers to improve I/O performance are inefficient on SMT processors, because they incur excessive context switches and spinning when workloads are waiting for I/O events. Such inefficiency makes it difficult to achieve high CPU throughput and high I/O throughput, which are required by typical workloads in clouds with both intensive I/O operations and heavy computation.

The paper proposes to use *context retention* as a key technique to improve I/O performance and efficiency on SMT processors. Context retention uses a hardware thread to hold the context of an I/O workload waiting for I/O events, such that overhead of context switches and spinning can be eliminated, and the workload can quickly respond to I/O events. Targeting virtualized clouds and x86 systems, the paper identifies the technical issues in implementing context retention in real systems, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling.

The paper designs vSMT-IO to implement the idea and the techniques. Extensive evaluation based on the prototype implementation in KVM and diverse real-world applications, such as DBMS, web servers, AI workload, and Hadoop jobs, shows that vSMT-IO can improve I/O throughput by up to 88.3% and CPU throughput by up to 123.1%.

1 Introduction

Simultaneous Multi-Threading (SMT), or Hyper-Threading (HT) on x86 processors, is widely enabled on most cloud infrastructures [1–4]. For example, in Amazon EC2 [1], virtual instances can have their virtual CPUs (vCPUs) run on dedicated hardware threads or time-share hardware threads. With SMT, multiple hardware threads share the same set of execution resources in each core, such as functional units and caches. Thus, when enabled, SMT can effectively improve resource utilization and system throughput.

On SMT processors, CPU schedulers are critical for achieving high performance. To make optimal scheduling decisions, they must fully consider and leverage the performance features of SMT processors, particularly the intensive resource sharing between hardware threads. For example, intensive study has concentrated on symbiotic scheduling algorithms, which co-schedule the threads that can fully utilize the hardware resources with minimal conflicts on each core [5–10].

Existing scheduling optimizations for SMT processors, including symbiotic scheduling and other enhancements in existing system software [11–13], mainly target computation-intensive workloads and aim to improve processor throughput. However, the techniques that can effectively and efficiently improve the performance of I/O-intensive workloads on SMT-enabled systems have not been paid enough attention. These techniques are particularly important when a system has both computation workloads and I/O workloads, and requires both high processor throughput and high I/O throughput.

To improve I/O workload performance, existing CPU schedulers generally use two techniques, polling [14–16] and boosting the priority of I/O workloads [17–19]. However, with these techniques, I/O workloads incur high overhead on SMT processors due to busy-looping and increased context switches, which can significantly reduce the performance of computation running on other hardware threads.

This problem is particularly significant and detrimental in clouds. In clouds, I/O workloads and computation workloads are usually consolidated on the same server to improve system utilization [17, 19–22]. At the same time, virtualization is dominantly used in clouds, which causes busy-looping and context switches to incur higher overhead, because extra operations must be carried out to deschedule and reschedule virtual CPUs, as we will show in §2.

To control the overhead of polling and I/O workload priority boosting, existing system designs make trade-offs between the efficiency and the effectiveness of these techniques, which undermine the performance of I/O workloads. For polling, existing systems usually incorporate a short timeout to keep the busy-looping brief. For priority boosting, it has been a long-standing dilemma to make I/O workloads preempting the running workloads promptly or with some extra delay; to resolve this dilemma, Linux uses a scheduling delay parameter (tunable, usually a few milliseconds) as a knob to trade-off I/O workloads responsiveness and the increased context switch overhead.

Instead of improving the effectiveness-efficiency trade-off, the paper seeks a fundamental solution to the above problem. The key is a technique that can effectively improve the performance of I/O workloads with high efficiency. Our solution is motivated by the hardware-based design for efficient blocking synchronization on SMT processors [23]. With the design, blocking synchronizations can be finished efficiently without busy-looping or context switches. Specifically, the design allows a thread blocked at a synchronization point to free all its resources for other hardware threads to use, except for its hardware context; thus, when the thread is unblocked, it can resume its execution in a few cycles.

Our solution targets virtualized clouds and x86 SMT processors. It is built on a hardware-based blocking mechanism for vCPUs, named **Context Retention**. Context retention is implemented with Intel `MONITOR/MWAIT` support [24]. With context retention, when a vCPU is waiting for an I/O event, its execution context can be held on a hardware thread without busy-looping involved; upon the I/O event, the vCPU can resume execution quickly without a context switch.

1.1 Technical Issues

While the rationale of the context retention mechanism is straightforward, maximizing its potential on improving performance needs to address three technical issues listed below. These issues arise mainly because context retention may be long time periods. Many I/O operations have long latencies in millisecond scale, and the latencies may further increase due to queueing/scheduling delays. To avoid context switches, the contexts of the vCPUs waiting for the finish of these operations need to be retained on hardware threads for the same amount of time.

First, uncontrolled context retention can diminish the benefits from simultaneous multithreading, because context retention reduces the number of active hardware threads on a core. This issue is particularly serious for x86 processors, which only implement 2-way SMT[1]. When a hardware thread is used for context retention, only one hardware thread remains for computation.

Second, context retention consumes the timeslice of an I/O workload, and thus reduces its timeslice available for computation. We found that, if not well controlled, context retention can even reduce the throughput of I/O workloads.

Third, due to context retention and burstiness of I/O operations [25], the resource demand of an I/O workload may vary dramatically on a hardware thread. This makes it a challenging task to improve processor throughput with existing symbiotic scheduling methods. To determine which workloads may make fast progress if scheduled on the same core, existing symbiotic scheduling methods periodically profile workload executions and make predictions based on the profiling results. Thus, these methods are effective only when the workload on each vCPU changes steadily. They must be substantially extended to handle I/O workloads.

1.2 Major Techniques

We implement our solution and address the above issues by designing the vSMT-IO scheduling framework. It has two major components. The **Long-Term Context Retention (LTCR)** mechanism is mainly to maximize I/O throughput with high efficiency. The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to maximize processor throughput.

The LTCR mechanism mainly addresses the first two issues identified in Section 1.1. It holds the context of the vCPU waiting for an I/O event on a hardware thread for an extended time period. If the expected I/O event happens in this period, the vCPU can quickly resume and respond to the event. Otherwise, the vCPU is descheduled. The maximum length of the time period is carefully adjusted in a way that both processor throughput and I/O throughput can be improved.

With LTCR, the context of an I/O workload can be held for as long as a few milliseconds, which is more than 10x longer than the busy-looping timeout used in system software (sub-millisecond) [14, 15]. This makes LTCR capable of dealing with relatively high I/O latencies, which are associated with slow I/O operations (e.g., HDD accesses and SSD writes) or caused by various system factors (e.g., queueing/scheduling delay and SSD block erase). In contrast, polling is used only when I/O workloads interact with low latency devices, e.g., local network and NVMe devices [16, 26].

The RASS algorithm mainly addresses the third issue identified in Section 1.1. On each core, it classifies the vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs. It uses one hardware thread for running CPU-bound vCPUs and the other hardware thread mainly for I/O-bound vCPUs. In this way, the computation on the CPU-bound vCPUs can overlap to the greatest extent with the context retention periods on the other hardware thread. This effectively improves processor throughput, since CPU-bound vCPUs can take advantage of the hardware resources released due to context retention to make fast progress. RASS schedules CPU-bound vCPUs on both hardware threads only when I/O-bound vCPUs are not ready to run. In this case, RASS selects CPU-bound vCPUs based on the symbiosis between vCPUs (i.e., how well the vCPUs can share the hardware resources and make progress when co-scheduled).

With RASS, the first two issues identified in Section 1.1 can

[1] Though some Xeon Phi processors implement 4-way SMT, the paper targets 2-way SMT x86 processors because of their overwhelming dominance in clouds.

be further mitigated. LTCR mainly targets long context retentions. It limits the lengths of context retentions to mitigate the resource underutilization they cause and reduce the timeslice they consume. However, it cannot deal with the issues caused by relatively short context retentions. For these context retentions, RASS mitigates the resource underutilization issue (the first issue in Section 1.1) by overlapping computation and context retention; to mitigate the second issue, it helps ensure the supply of timeslice to I/O-bound vCPUs by running them on dedicated hardware threads with high priorities.

The paper makes the following contributions. First, the paper identifies the efficiency issues in existing CPU schedulers when they are used to improve I/O performance on SMT-enabled systems, and proposes a novel idea, context retention, to improve efficiency. Second, it identifies the issues in implementing the idea, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling. Third, targeting virtualized clouds and x86 processors, the paper designs vSMT-IO to implement the idea and the techniques, and builds a system prototype based on KVM [27]. Forth, it has evaluated vSMT-IO with extensive experiments and a diverse set of 18 programs, including DBMS, web servers, AI workloads, and Hadoop jobs, and compared the performance of vSMT-IO with the vanilla system and widely-adopted enhancements. The experiments show that vSMT-IO can improve I/O throughput by up to 88.3% and processor throughput by up to 123.1%.

2 Background and Motivation

Targeting virtualized clouds, this section demonstrates the efficiency issues of existing schedulers in improving I/O performance on SMT-enabled systems. It first introduces these techniques, and experimentally verifies their inefficiency and the caused performance degradation (§2.1). Then, it explains why the issues are serious on virtualized platforms (§2.2).

2.1 Inefficient I/O-Improving Techniques

I/O-intensive applications are usually driven by I/O events. A pattern repeated in their executions is waiting for I/O events (e.g., queries received from network, or data read from disks), processing I/O events, and generating new I/O requests (e.g., responses to queries, or more disk reads). Thus, high I/O performance not only depends on fast and well-managed I/O devices to quickly respond to I/O requests. It also depends on the applications to promptly respond to various I/O events, such that new I/O requests can be generated and issued to I/O devices quickly.

Thus, CPU schedulers play an important role in improving I/O performance. To increase the responsiveness of I/O workloads to I/O events, existing schedulers use two general techniques — polling for low-latency I/O events and priority boosting for high-latency I/O events. With polling, an I/O workload waiting for an I/O event enters a busy loop (implemented with PAUSE on x86 processors) with a pre-set timeout. The workload keeps looping before it is interrupted upon the expected I/O event or is descheduled due to timeout. Thus, polling allows a workload to respond to I/O events with a minimal delay before timeouts. With priority boosting, upon an I/O event, the priority of the I/O workload is boosted, such that it can quickly preempt a running workload to respond to the I/O event.

On virtualized platforms, I/O workloads run on vCPUs; and vCPU scheduling becomes a key component affecting I/O performance. For vCPUs, polling may be implemented in guest OS kernel [28]. However, busy-looping in guest OS causes unnecessary VM_EXITs and extra overhead on x86 processors when Pause Loop Exiting (PLE) is enabled. Thus, recent designs (e.g., HALT-Polling [15]) usually implement polling at the VMM level. Priority boosting may be implemented by adjusting priorities explicitly [17] or by implicitly associating priorities with CPU time consumption. For example, Linux/KVM allows the vCPUs with lower CPU time consumption (e.g., I/O-bound vCPUs) to preempt the vCPUs with higher CPU time consumption [17, 29].

Though polling and priority boosting can improve the performance of I/O workloads, they are inefficient on SMT processors. The operations associated with these techniques, busy-looping and context switches, waste the hardware resource that can be otherwise utilized by the computation on other hardware threads. Thus, the inefficiency may not be an issue when a system has only I/O workloads; but it becomes detrimental when I/O workloads are consolidated with computation workloads. Efficiency can be improved by making these techniques less aggressive, e.g., enforcing a shorter timeout for polling. However, this sacrifices the effectiveness of these techniques and I/O performance.

We illustrate the inefficiency issue with polling and priority boosting using the experiments with two combinations of applications, `Sockperf` with `Matmul`, and `Redis` with `PageRank`. `Sockperf` and `Redis` are I/O-bound. `Matmul` and `PageRank` are CPU-bound. We run each combination on a 24-core server (48 hyperthreads) with each application running in a 48-vCPU VM. This results in 2 vCPUs on each hyperthread. The VMs are managed by KVM/Linux. Detailed server/VMs configurations and application descriptions can be found in §6.

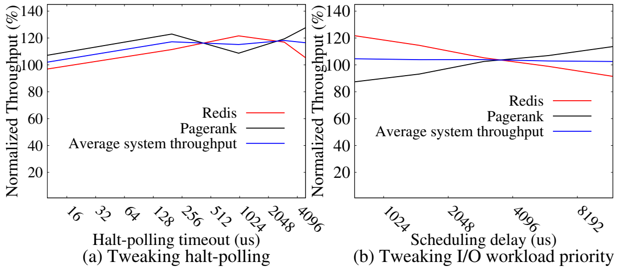
To illustrate the inefficiency issue on a well-tuned system with high efficiency, we have enhanced the HALT-Polling implementation in KVM. The enhancement makes HALT-Polling more effective, so as to further reduce context switches between vCPUs and make vCPUs more responsive to I/O events. Specifically, with the "vanilla" implementation, an idle vCPU is not allowed to perform HALT-Polling when there is another vCPU ready to run on the same hyperthread. The enhancement removes this restriction. It also increases the maximum timeout that is allowed in HALT-Polling. (HALT-Polling adjusts timeout value dynamically between 0 and

workloads	KVM w/ enhanced HALT-Polling			VSMT-IO		
	vCPU switches	spinning time	perf. imprv.	vCPU switches	spinning time	perf. imprv.
`Sockperf`	12.5K	40.1%	16.1%	3.3K	-	56.5%
`Matmul`			8.2%			57.4%
`Redis`	43.9K	27.5%	8.4%	15.1K	-	88.3%
`PageRank`			7.7%			123.1%

Table 1: ***Existing techniques handling I/O workloads incur frequent vCPU switches and massive spinning, and are inefficient on SMT processors.*** **"vCPU switches" are counts of context switches between vCPUs every second in the server. The performance improvements are relative to "vanilla" KVM.**

a maximum value.) The enhancement improves the performance of the applications by 7.7% ∼ 16.1%.

As shown in Table 1, both application combinations incur frequent vCPU switches. For example, `Redis` and `PageRank` incur a vCPU switch about every 1 millisecond on each hyperthread. At the same time, a substantial portion of CPU time is spent by polling (e.g., 40.1% for `Sockperf` and `Matmul`). vCPU switches and such massive polling inevitably degrade performance, as we will show later.



(a) Tweaking halt-polling

(b) Tweaking I/O workload priority

Figure 1: ***Tweaking existing techniques for scheduling I/O workload cannot substantially improve performance.*** **(The throughputs are normalized to those with vanilla KVM.)**

The performance advantage of the enhanced HALT-Polling is achieved by increasing polling to reduce costly vCPU switches. This demonstrates some potential to tweak existing designs. However, to improve performance significantly, major changes must be made. To illustrate this, Figure 1 shows how the performance of `Redis` and `PageRank` changes when tweaking the key parameters of polling and priority boosting. We first tweak the timeout used in HALT-Polling and vary it from 10 microseconds to 5 milliseconds. Figure 1(a) shows that increasing timeout only slightly improves performance when timeout value is small. However, the performance improvement of these two applications hits a plateau at about 10% after the timeout value reaches 200 microseconds.

Then, we adjust the scheduling delay parameter in Linux. The parameter controls the delay between a vCPU being woken up upon an I/O event and the vCPU preempting another vCPU. Thus, increasing the parameter essentially reduces the priority of I/O-bound vCPUs and reduces vCPU switches. As Figure 1(b) shows, the average performance barely changes; and increasing this parameter is basically sacrificing I/O performance for higher processor throughputs.

The aim of VSMT-IO is to substantially reduce the overhead caused by spinning and vCPU switches. The reduced overhead improves the performance of computation workloads. As shown in Table 1, reducing more than 2/3 of vCPU switches and eliminating spinning lead to significant performance improvement to `PageRank` (123.1% relative to vanilla KVM or 107.1% relative to enhanced KVM). More importantly, the performance improvement of computation workload is not at the cost of I/O performance. With VSMT-IO, the throughput of `Redis` is increased by 88.3% over vanilla KVM or 73.7% over enhanced KVM. The system I/O throughput is also increased by 75.1% over enhanced KVM.

2.2 Overhead of Polling and Context Switches

Existing techniques for improving I/O performance are inefficient on SMT processors, because context switches and polling waste the resource that can be otherwise utilized by the computation on other hardware threads. Targeting virtualized clouds, this subsection highlights the overhead of these operations with experiments and explains how such high overhead is incurred.

Hyperthread 1	Hyperthread 2	Relative performance
-	`Matmul`	100%
vCPUs Switches	`Matmul`	32%
HALT-Polling	`Matmul`	73%

Table 2: ***vCPU switches and HALT-Polling on a hyperthread slow down the computation on the other hyperthread.***

In the experiments, we run a `Matmul` thread on a hyperthread. Then, on the other hyperthread, we make two vCPUs switch back and forth or make a vCPU repeat the HALT-Polling loop. We check how the performance of `Matmul` is impacted by these operations.

The experiments show that vCPU switches slow down `Matmul` by about 70%, and HALT-Polling slows it down by about 30% (Table 2). While the slowdowns explain the inefficiency of polling and priority boosting techniques, we were surprised at these slowdowns. We expected the slowdown caused by vCPU switches to be around 50%, because there are two streams of instructions compete for CPU resource on the hyperthreads, and expected the slowdown caused by HALT-Polling to be minimal, because PAUSE instruction is designed to consume minimal resource.

We have diagnosed the slowdowns. vCPU switches cause large slowdowns mainly because the L1 data cache shared by both hyperthreads needs to be flushed during vCPU switches to address the *L1 Terminal Fault* problem [30, 31]. Other costly operations, including TLB flush [32], handling rescheduling IPIs [33], and the execution of scheduling algorithm, also contribute to the performance impact incurred by vCPU switches. The slowdown caused by HALT-Polling is larger than expected because the operations other than PAUSE are executed. HALT-Polling is implemented in the VMM. Thus, VM_EXIT is incurred when a vCPU enters HALT-Polling. VM_EXITs are costly operations [34]. During the polling, the instructions controlling the busy-loop are executed repeatedly. They are also more costly than PAUSE.

3 Basic Idea and Technical Issues

As Section 2 shows, polling and priority boosting incur high overhead on SMT processors; tweaking these techniques yields only marginal performance improvements. This requires that a new and efficient technique be developed to handle I/O workloads.

On a SMT processor, an efficient technique must consume minimal hardware resources. In a scheduling technique for improving I/O performance, two factors determine its hardware resource consumption. One is how to handle an I/O workload while it is waiting for the completion of an I/O operation. The other is how to quickly resume the execution of the I/O workload upon the completion of the expected I/O operation. Polling and priority boosting each concentrate on reducing the resource consumption of only one factor, but at the cost of high resource consumption in the other factor. Our solution aims to minimize the resource consumption of both factors.

Our solution leverages two features of SMT processors: 1) hardware-based blocking support, and 2) intense resource sharing between hardware threads. With these features, we implement a **Context Retention** mechanism for vCPUs. While a vCPU is waiting for the completion of I/O operations, it can "block" itself on a hardware thread, and release all its resources for other hardware threads to use, except for its hardware context. This minimizes the resource consumption required by waiting for the completion of I/O operations. With the hardware context, the vCPU can be quickly "unblocked" without context switches upon the completion of the I/O operations. This minimizes the resource consumption required to quickly resume the execution of I/O workloads. Table 3 summarizes the benefits of context retention from the perspectives of both I/O workloads and computation workloads.

	Benefit	**Overhead**
I/O	better responsiveness	timeslice charged for context retention
Computation	extra resources from reduced context switches and polling	resource underutilization

Table 3: ***A summary of benefit and overhead of context retention.***

Though context retention consumes minimal hardware resources, it does incur some overhead, which are as summarized in Table 3 and must be reduced for better efficiency. From the perspective of computation workloads, because not all the hardware threads can be used for computation, the overhead is reflected by resource underutilization. Given that a x86 core has two hyperthreads, to avoid low utilization, one must be doing computation while the other does context retention. Even with this arrangement, full utilization may not be achieved.

From the perspective of I/O workloads, they are charged for vCPU usage while they retain contexts; so only short context retention periods are cheaper than descheduling and rescheduling vCPUs; but longer retention periods are not. This problem can be illustrated by the performance of I/O workload `Redis` in Figure 1(a). Increasing HALT-Polling timeout improves the performance of `Redis` when the timeout value is low. However, after the timeout exceeds 0.5 millisecond, further increasing the timeout degrades its performance. This is because, with a longer timeout, polling consumes more timeslice and reduces the timeslice available to the computation in `Redis`. Though polling is used in this experiment, if polling is replaced with context retention, the performance trend would be similar.

For the above overhead issues, a natural solution is to control the maximum length of context retention, such that extended context retention periods will not cause high overhead. However, this solution cannot deal with the overhead of the context retention periods that are relatively short. Reducing this overhead requires some enhancement in vCPU scheduling. For example, resource underutilization can be mitigated by scheduling a resource-demanding vCPU on a hyperthread when context retention is in progress on the other hyperthread; the vCPUs with much timeslice consumed by context retention can be compensated with extra timeslice.

In addition to the overhead issues, context retention also creates some challenge on the integration of symbiotic scheduling methods, which are needed for improving CPU performance. The key of symbiotic scheduling is to estimate how well a group of workloads can corun on a SMT core (i.e., symbiosis level) [6–9, 35]. This is achieved by monitoring workload executions periodically. For instance, SOS (Sample, Optimize and Symbiosis) [5] samples workload executions periodically in sample phases to determine their symbiosis levels, and preferentially coschedules tasks with the highest symbiosis levels in symbiosis phases. Thus, existing symbiotic scheduling methods require that the resource demand of a workload change steadily during its execution. Due to context retention and burstiness of I/O operations [25], the resource demand of an I/O workload changes dramatically during its execution on a vCPU. Existing symbiotic scheduling methods cannot handle such workloads. This issue may be addressed by coscheduling I/O workloads with computation workloads, such that symbiosis levels can be lifted by overlapping context retention with resource-demanding computation. Existing symbiotic scheduling methods can still be used to handle computation workloads.

4 vSMT-IO Design

We implement our idea and address the technical issues in vSMT-IO. In this section, we present the overall architecture of vSMT-IO and its major components.

4.1 Overview

Figure 2 shows the overall architecture of vSMT-IO. vSMT-IO incorporates four major components:

• The **Long Term Context Retention (LTCR)** mechanism on each core implements context retention. To prevent extended context retention periods causing high overhead (re-

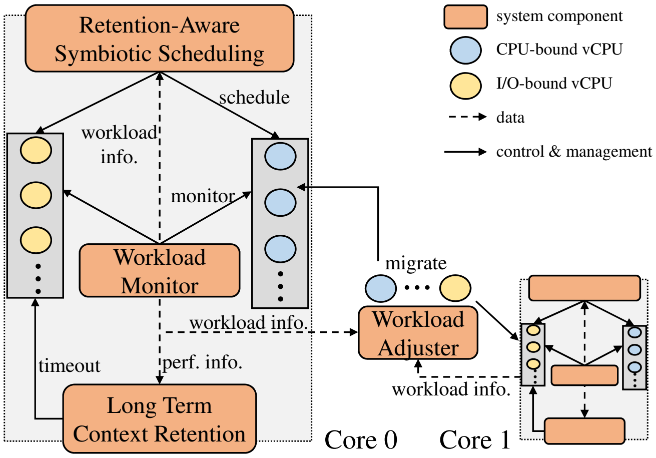


Figure 2: vSMT-IO *Architecture.* Key components are in orange.

source underutilization and timeslice consumption), it enforces a context retention timeout, and dynamically adjusts the timeout value.

- The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to increase the symbiosis levels of the vCPUs running on the hypertheads in each core. To achieve this, RASS classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules CPU-bound vCPUs on a hyperthread and I/O-bound vCPUs on the other hyperthread. CPU-bound vCPUs run on both hyperthreads only when I/O-bound vCPUs are not ready to run. In this way, the resource-demanding computation on CPU-bound vCPUs can overlap to the greatest extent with the resource-conserving context retention periods on I/O-bound vCPUs. Increased symbiosis levels improve CPU performance and reduce the overhead of context retentions. At the same time, using a dedicated hyperthread for I/O-bound vCPUs allows them to use extra CPU time as a "compensation" for the timeslice charged in context retention periods, and further prevents them from being unfairly penalized.
- The **Workload Monitor** on each core monitors vCPU executions. It characterizes the workloads on the vCPUs and measures performance. It provides workload information for RASS to classify and schedule vCPUs and for the workload adjuster introduced below to adjust the workloads between cores. It provides performance information for LTCR to adjust the timeout value.
- The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. The **Workload Adjuster** supplements RASS. It adjusts the workloads on each core to maintain their heterogeneity by migrating vCPUs between cores.

4.2 Long Term Context Retention (LTCR)

On x86 processors, we implement vCPU context retention with the `MONITOR/MWAIT` support. Specifically, to wait for an I/O event, a vCPU calls a `MWAIT` instruction paired with a `MONITOR` instruction that specifies a memory location in guest OS. The `MWAIT` instruction "blocks" the vCPU and keeps its context on the hyperthread. With the `MONITOR/MWAIT` support, the `MWAIT` instruction ends automatically when the content at the memory location is updated or an interrupt is directed to the hyperthread. Since both I/O events and timeouts can be notified with interrupts, we choose to use interrupts to stop `MWAIT`. To prevent `MWAIT` from being terminated by memory writes prematurely, we set the memory location used in `MONITOR` read-only.

Algorithm 1 Context Retention Timeout Adjustment

```
$T_d$: desired timeout value; $T_e$: effective timeout value; $T_{init}$: initial time-
  out value; $P$: time period between two adjustments
$T_d \leftarrow T_{init}$
while true do
    $T_e \leftarrow T_d$, collect performance data for a time period of $P$
    if TESTTIMEOUT($T_d$ * 1.1) then
        $T_d \leftarrow T_d * 1.1$; continue
    else
        $T_e \leftarrow T_d$, collect performance data for a time period of $P$
    end if
    if TESTTIMEOUT($T_d$ * 0.9) then
        $T_d \leftarrow T_d * 0.9$; continue
    end if
end while
function TESTTIMEOUT($T$)
    $T_e \leftarrow T$, collect performance data for a time period of $P$
    $S_{cpu} \leftarrow$ average speed-up of CPU-bound vCPUs
    $S_{io} \leftarrow$ average speed-up of I/O bound vCPUs
    if $S_{cpu} > 1$ and $S_{io} > 1$ then return true; end if
    return false
end function
```

The context retention timeout is to balance the cost and benefit of context rentention. Based on the summary in Table 3, for I/O workloads, lengthening a context retention is always a gain when it consumes less timeslice than descheduling and then rescheduling a vCPU. For computation workloads, context retention is rewarding when the amount of resource saved by reducing context switches and polling exceeds the amount of resource that cannot be utilized due to context retention. In the cases where one hyperthread does computation and the other hyperthread does context retention, context retention is always rewarding if it is not longer than the time spent on descheduling and then rescheduling a vCPU, based on the measurements shown in Table 2. Thus, context retention timeout can be set to be at least the time required by descheduling and rescheduling a vCPU. Then, longer timeouts can be tested.

LTCR uses algorithm 1 to adjust the context retention timeout periodically. The algorithm slightly increases or decreases the timeout value, checks whether performance is improved with the new value, and keeps the new value if it is. The algorithm uses the vCPU performance information collected by the workload monitor to determine whether performance is improved. Specifically, it uses IPC (instruction per cycle) to measure the performance of CPU-bound vCPUs, and uses the frequency of context retentions (i.e., number of context retentions per second) to measure the performance of I/O-bound vCPUs. Then, the algorithm calculates a speed-up for each vCPU. A speed-up value greater than 1 indicates that the performance of the vCPU has been improved with the new timeout value. It averages the speed-up values of CPU-bound vCPUs, and averages the speed-up values of I/O-bound vCPUs. The algorithm determines that the performance is improved and the new timeout value should be kept only if both average values are greater than 1.

4.3 Retention Aware Symbiotic Scheduling (RASS)

RASS schedules the vCPUs on each core with the main aim of maximizing the computation throughput of the core. This is achieved by increasing the symbiosis levels of the vCPUs running on the hypertheads. RASS combines two methods. One is *unbalanced scheduling* that maximizes the overlapping between resource-demanding computation and resource-conserving context retention periods (Section 4.3.1). The other is *symbiotic scheduling based-on cycle accounting* to select CPU-bound vCPUs with high symbiosis levels when both hardware threads need to run CPU-bound vCPUs (Section 4.3.2).

4.3.1 Unbalanced Scheduling

Unbalanced scheduling classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules them on paired hyperthreads (See Figure 3). The classification is based on how much time each vCPU spends on context retention. Specifically, for each vCPU, a context retention rate is calculated and updated periodically. It is the ratio between the time spent on context retention in last time period and the period length. When a new period begins, the vCPUs are ranked based on their context retention rates. The vCPUs with higher context retention rates are considered to be I/O-bound, and the rest are CPU-bound.

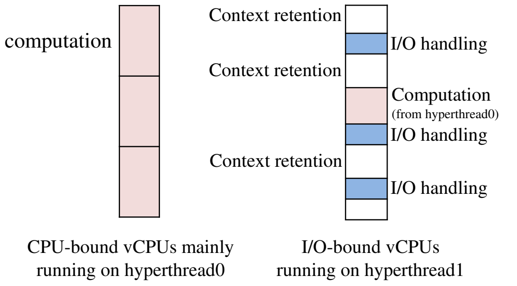


Figure 3: Computation and context retention are distributed to different hyperthreads with unbalanced scheduling.

When the hyperthread running I/O-bound vCPUs is idle, a CPU-bound vCPU is selected based on the symbiosis level (Section 4.3.2) and migrated to this hyperthread. This is to improve the utilization of CPU hardware to further increase CPU performance. The CPU-bound vCPU can only run with a priority lower than the I/O-bound vCPUs. It is preempted and migrated back when an I/O-bound vCPU becomes ready to run. This is to prevent the CPU-bound vCPU from blocking I/O-bound vCPUs and degrading I/O performance.

Unbalanced scheduling assumes that each vCPU has been attached with a weight, e.g., that used in Linux Completely Fair Scheduler (CFS). When classifying the vCPUs, it tries to balance the total weight of CPU-bound vCPUs and the total weight of I/O-bound vCPUs, and make them roughly equal. This is mainly to balance the load on the hyperthreads and reduce the migration of CPU-bound vCPUs.

The compensation to I/O-bound vCPUs for the timeslice consumed by context retentions can also be implemented by adjusting the weights of vCPUs. For example, the weights of the vCPUs can be increased based on their context retention rates. For the vCPUs that spend more time on context retentions than other vCPUs, their weights are increased by larger percentages. In this way, fewer vCPUs are classified as I/O-bound, and share the same hyperthread. However, we found that this adjustment is not necessary in most cases. The main reason is that I/O-bound vCPUs usually have low CPU utilization. Thus, even with context retention, some I/O-bound vCPUs still cannot fully consume their timeslice. Other I/O-bound vCPUs that need more timeslice acquire automatically extra timeslice as compensation. This is because the scheduler is work-conserving, and I/O-bound vCPUs have higher priority than CPU-bound vCPUs on the hyperthread and are supplied with extra timeslice first.

4.3.2 Symbiotic Scheduling Based on Cycle Accounting

When both hyperthreads need to run CPU-bound vCPUs, the symbiosis levels between vCPUs must be considered. RASS determines the symbiosis levels using the cycle accounting technique [36–39]. It is a symbiotic scheduling technique for threads. We only adapt its method that estimates the symbiosis levels between threads and use it on vCPUs.

We select this technique because of its high practicality. To estimate the symbiosis levels between threads, it samples and characterizes each individual thread, and inputs the characterization into an interference estimation model. Compared to SOS (Sample, Optimize and Symbiosis), which samples the execution of possible thread combinations [5], the cycle accounting technique has a much lower complexity ($O(n)$ vs. $O(n^2)$) and thus higher practicality.

The cycle accounting technique uses three parameters, which are the components of the CPI (cycler per instruction), to characterize a thread. The **b**ase component (**B**) is the number of cycles used to finish an instruction when all the required hardware resource and data are locally available; the **m**iss component (**M**) is the number of cycles used to handle misses (e.g., cache misses and TLB misses); the **w**aiting component (**W**) is the number of cycles waiting for hardware resource to become available. The CPI value is roughly the sum of B, M, and W.

When the parameters of a thread are being measured, the cycle accounting technique requires that the thread run alone on the core without any computation on the other hyperthread so as to eliminate interference. This incurs non-trivial overhead. To reduce this overhead, we take advantage of context retentions, and measure the parameters of a CPU-bound vCPU when it is running on a hyperthread and context retention is in progress in the other hyperthread. We obtain the base component, the miss component, and the CPI of the vCPU using hardware counters, and calculate the waiting component from this.

4.4 Workload Adjuster

The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. Its performance advantage may diminish when workloads become homogeneous due to factors, such as load balancing and phase changes in workloads. The workload adjuster is designed to maintain the workload heterogeneity on each core.

The workload adjuster measures workload heterogeneity and characterizes the overall workload type by calculating the standard deviation and the average value of vCPU context retention rates. If a group of vCPUs have a small deviation value, their workloads are generally homogeneous; if the average context retention rate of a group of vCPUs is very high, these vCPUs are likely to be I/O-bound; if the average rate is very low, the vCPUs are likely to be CPU-bound. The workload adjuster calculates these values for each core, and updates them periodically to detect the need for workload adjustment. When the standard deviation drops below a pre-set threshold, workload adjustment starts.

To adjust the workloads, the adjuster finds the core with the smallest deviation. Then, based on the average context retention rate of the core (e.g., a very small average value of CPU-bound vCPUs), the adjuster searches for another core, which is dominated by the other type of vCPUs (e.g., I/O-bound vCPUs). The search is done by examining the average context contention rates of other cores. The desired core is the one with the average context contention rate that differs from the former average rate by the largest degree (e.g., a very large average value of I/O-bound vCPUs). After a such core is found, the adjuster ranks the vCPUs based on their context retention rates on each of these two cores, selects the vCPU ranked in the middle on each core, and swaps the two vCPUs.

5 Implementation Details

We have implemented a prototype of vSMT-IO based on Linux/KVM. We added/modified about 1300 lines of source code mainly in KVM kernel modules and Linux CFS [2]. The workload monitor and the long-term context retention (LTCR) components are mainly implemented in a KVM kernel module by changing *kvm_main.c*. In LTCR, the context retention mechanism needs to be implemented in guest OS to minimize overhead. Though it can be implemented as an idle driver kernel module [40], we choose to directly change the idle loop in *idle.c* to simplify the implementation. Context retention is implemented with a loop, which repeatedly calls `MONITOR`, `MWAIT`, and the *need_sched()* function of Linux kernel. It is inserted at the beginning of each iteration of the idle loop. Implementing context retention with a loop is to prevent it from being terminated prematurely by irrelevant interrupts. The loop terminates when a thread becomes "ready" on the vCPU (fulfilled with the *need_sched()* call). Thus, context retention can finish upon the expected I/O event. The loop also ends if a timer interrupt "marking" the timeout of the context retention is received by the vCPU. To differentiate this interrupt from regular timer interrupts, we change the two unused bits in the VM execution control register, and use them as a timeout flag.

Retention aware symbiotic scheduling and workload adjuster are implemented based on Linux CFS in *fair.c* and *core.c*. Thus, the original scheduling and load balancing policies implemented in CFS are followed in most cases, e.g., when deciding which I/O-bound vCPU is the next to run on a hyperthread. However, when deciding which CPU-bound vCPU is the next to run, the symbiotic scheduling policy in RASS and the fairness based scheduling policy in CFS have different objectives, and thus may decide to select different vCPUs. To coordinate these different objectives, our implementation let Linux CFS select a few vCPUs based on its policies. Then, among these vCPUs, RASS selects a vCPU based on symbiosis.

6 Performance Evaluation

With the prototype implementation, we have evaluated vSMT-IO extensively with a diverse set of workloads. The objectives of the evaluation are four-fold: 1) to show that vSMT-IO can improve I/O performance with high efficiency and benefit both I/O workload and computation workload, 2) to verify the effectiveness of the major techniques used in vSMT-IO, 3) to understand the performance advantages of vSMT-IO across diverse workload mixtures and different scenarios, and 4) to evaluate the overhead of vSMT-IO.

6.1 Experiment Settings

Our evaluation was done on a DELL^{TM} PowerEdge^{TM} R430 server with two 2.60GHz Intel Xeon E5-2690 processors (two NUMA zones), 64GB of DRAM, a 1TB HDD, and an Intel I350 Gigabit NIC. Each processor has 12 physical cores, and each physical core has two hyperthreads. With KVM, we built four VMs, each with 24 vCPUs and 16GB memory. Both the host OS and guest OS are Ubuntu Linux 18.04 with kernel updated to 5.3.1. We test vSMT-IO with a large and diverse set of workloads generated by typical applications from different domains, as summarized in Table 4. In the experiments, each VM encapsulates one workload.

We test vSMT-IO under two settings. Under the first setting, we launch two VMs; thus each vCPU has a dedicated

[2] Source code can be found at https://github.com/vSMT-IO/vSMT-IO.

App.	Workload Description
Redis	Serve requests (randomly chosen keys, 50% SET, 50% GET) [42].
HDFS	Read 10GB data sequentially with HDFS TestDFSIO [43].
Hadoop	TeraSort with Hadoop [43].
HBase	Read and update records sequentially with YCSB [44].
MySQL	OLTP workload generated by SysBench for MySQL [45].
Nginx	Serve web requests generated by ApacheBench [46].
ClamAV	Virus scan a large file set with clamscan [47].
RocksDB	Serve requests (randomly chosen keys, 50% SET, 50% GET) [48].
PgSQL	TPC-B-like workload generated by PgBench [49].
Spark	PageRank and Kmeans algorithms in Spark [50].
DBT1	TPC-W-like workload [51].
XGBoost	Four AI algorithms included in XGBoost [52] system.
Matmul	Multiply two 8000x8000 matrices of integers.
Sockperf	TCP ping-pong test with Sockperf [53].

Table 4: Benchmark applications used to test vSMT-IO.

hyperthread. We compare vSMT-IO against three competing solutions: 1) Blocking, which immediately deschedules the vCPUs waiting for I/O events, and is implemented by disabling HALT-Polling in KVM; 2) Polling, which is implemented by booting guest OS with parameter *idle=poll* configured [41] (timeout is not enforced for best I/O performance); and 3) HALT-Polling implemented in KVM, which combines polling and priority boosting techniques.

Under the second setting, we launch four VMs; thus, each hyperthread is time-shared by two vCPUs. Without a timeout, Polling is not a choice for improving I/O performance under this setting. Thus, we compare vSMT-IO against 1) vanilla KVM, which uses priority boosting to improve I/O performance, because HALT-Polling implemented in vanilla KVM is inactive under this setting, and 2) HALT-Polling enhanced to support time-sharing (described in Section 2.1).

We measure the throughputs of the workloads. We also collect response times if the workloads report them. The performance measurements may vary significantly across different workloads. When we present them in figures, for clarity, we normalize them against those of Blocking under the first setting and priority boosting (i.e., vanilla KVM) under the second setting.

6.2 One vCPU on Each Hyperthread

Under the first setting, I/O workloads can achieve the best performance with Polling. We want to compare the effectivenss of vSMT-IO on improving I/O performance against that of Polling by comparing the performance of I/O workloads managed with these two solutions. Without a timeout, Polling incurs high overhead on SMT processors, and degrades the performance of other workloads on the processors. Blocking and HALT-Polling are more efficient solutions than Polling under this setting. We want to compare the efficiency of vSMT-IO against that of Blocking and HALT-Polling by comparing the performance of computation workloads when they are collocated with I/O workloads managed with these three solutions.

With the above objectives, we launch two VMs. We run Matmul in one VM, which is computation-intensive, and run an I/O-intensive benchmark in the other VM. Figure 4 shows the normalized throughputs of Matmul and eight I/O-intensive benchmarks selected to co-run with Matmul. Note that the performance with Blocking is shown with the flat line at 100%.

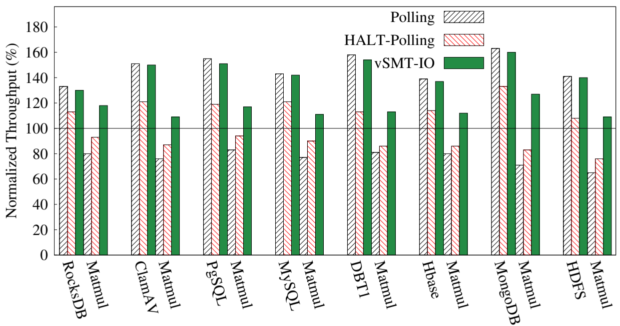


Figure 4: Throughputs of Matmul and eight I/O-intensive benchmarks when Matmul is collocated with each of the benchmarks in two VMs. Each vCPU runs on a dedicated hyperthread. Throughputs are normalized to those of Blocking.

With vSMT-IO, the I/O-intensive benchmarks achieve similar performance as they do with Polling. The largest difference is with DBT1, 4.1%. This is because DBT1 incurs a large number of random accesses to the HDD, which have long latencies exceeding the timeout value used in LTCR. On average, the I/O intensive benchmarks are only 2.3% slower with vSMT-IO. This shows that vSMT-IO is highly effective on improving I/O performance.

The high effectiveness of vSMT-IO is achieved with high efficiency. This is reflected by Matmul achieving higher performance with vSMT-IO consistently in all the experiments than it with the other three solutions. On average, with vSMT-IO the performance of Matmul is 37.9%, 14.5%, and 27.6% higher than it with Polling, Blocking, and HALT-Polling, respectively.

6.3 Multiple vCPUs Time-Sharing a Hyperthread

With multiple vCPUs on each hyperthread, context switches are usually incurred when improving I/O performance. It becomes more difficult for I/O-improving solutions to maintain high efficiency. We want to know to what extent the effectiveness and efficiency of vSMT-IO can be maintained. At the same time, vSMT-IO can be fully exercised under this setting. We want to verify the effectiveness of the major techniques in vSMT-IO.

In the experiments, we launch four VMs. On two of the VMs, we run two instances of the same benchmark, which is computation-intensive, e.g., Nginx, or AI algorithms in XGBoost. On the other two VMs, we run two instances of another benchmark, which is I/O-intensive, e.g., web server, or file server.

Figure 5 shows the normalized throughputs for eight benchmark pairs. In each pair, the first benchmark is I/O intensive, and the second benchmark is computation intensive. The enhanced HALT-Polling can effectively improve the throughputs of I/O-intensive benchmarks, because polling can "absorb" some context switches caused by I/O operations. Compared to vanilla KVM, the throughputs of I/O intensive benchmarks are increased by 36.9% on average. However, polling consumes CPU resources and may degrade the performance of other workloads (e.g., Nginx and Regression). Because the length of polling is carefully controlled in HALT-Polling, on average the throughputs of computation-intensive benchmarks are similar to those with vanilla KVM.

Compared to enhanced HALT-Polling, vSMT-IO can more effectively improve the throughputs of I/O-intensive benchmarks. On average, their throughputs are 29.5% higher than those with enhanced HALT-Polling. More importantly, this is achieved by improving the throughputs of computation-intensive workloads at the same time. On average, the throughputs of computation-intensive workloads with vSMT-IO are 22.8% and 18.4% higher than those with enhanced HALT-Polling and vanilla KVM, respectively.

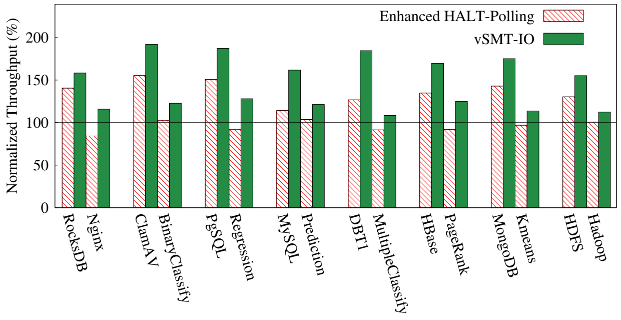


Figure 5: Throughputs of eight pairs of benchmarks. Each benchmark has two instances running on two VMs. Each hyperthread is time-shared by 2 vCPUs. Throughputs are normalized to those with vanilla KVM. Benchmarks BinaryClassify, MultipleClassify, Regression and Prediction are AI algorithms in XGBoost [52, 54].

The results in Figure 5 confirm that vSMT-IO can maintain its effectiveness and efficiency when each hyperthread is time-shared by vCPUs. To further investigate how the throughputs are improved with vSMT-IO, we collect the frequencies of vCPU switches (shown in Table 5) and profile the workload on the hyperthreads for I/O-bound vCPUs (results shown in Table 6).

The effectiveness of vSMT-IO on improving I/O performance relies on context retentions holding vCPU contexts on hyperthreads (the LTCR component). It is reflected by reduced context switches. As shown in Table 5, vSMT-IO can reduce vCPU switches significantly by up to 95% (80% on average). As a comparison, enhanced HALT-Polling can only reduce vCPU switches by at most 51% (32% on average). This explains the superiority of vSMT-IO over HALT-Polling.

Benchmark Pairs	Number of vCPU Switches Per Second		
	Vallina KVM	Enhanced HALT-Polling	vSMT-IO
(RocksDB,Nginx)	29.3k	15.2k	1.9k
(ClamAV,BinaryClassify)	11.8k	8.7k	3.2k
(PgSQL,Regression)	9.5k	8.0k	2.8k
(MySQL,Prediction)	11.5k	9.3k	4.5k
(DBT1,MultipleClassify)	61.3k	29.5k	3.9k
(HBase,PageRank)	23.4k	12.3k	3.9k
(MongoDB,Kmeans)	33.3k	20.8k	9.3k
(HDFS,Hadoop)	34.0k	30.6k	1.7k

Table 5: The number of vCPU switches is substantially reduced with vSMT-IO for the eight benchmark pairs.

Benchmark Pairs	Context Retentions	I/O Workload	Computation Workload
(RocksDB,Nginx)	28.1%	34.3%	37.6%
(ClamAV,BinaryClassify)	39.8%	31.6%	28.6%
(PgSQL,Regression)	42.3%	19.2%	38.5%
(MySQL,Prediction)	30.0%	33.5%	36.5%
(DBT1,MultipleClassify)	32.7%	54.4%	12.9%
(HBase,PageRank)	53.9%	31.9%	14.2%
(MongoDB,Kmeans)	34.4%	45.3%	20.3%
(HDFS,Hadoop)	33.0%	45.2%	21.8%

Table 6: Time (percentage) spent by context retentions, I/O-bound vCPU, and CPU-bound vCPU on the hyperthreads for I/O-bound vCPUs.

The high efficiency of vSMT-IO comes partially from its capability to reduce vCPU switches. It also comes from LTCR and RASS controlling the overhead incurred by context retentions. While the effectiveness of RASS on controlling the overhead is self-evident, the effectiveness of LTCR can be confirmed with the results shown in Table 6. LTCR limits the context retention lengths to prevent high overhead. As a result, on the hyperthreads for I/O-bound vCPUs, for most benchmark pairs, the time spent on context retentions is less than 40%. With context retention lengths well controlled, more than 20% of the CPU time on these hyperthreads can be used by CPU-bound vCPUs to improve CPU throughput.

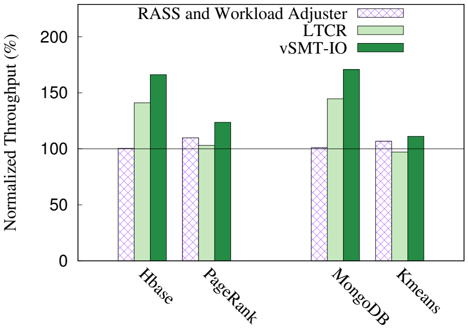


Figure 6: Normalized throughputs (relative to those achieved with vanilla KVM) of two pairs of benchmarks when LTCR and RASS are enabled separately.

To understand how the two major techniques in vSMT-IO, LTCR and RASS, improve performance, we enable these techniques separately, and show the performance of two pairs of benchmarks, HBase with PageRank, and MongoDB with

Kmeans, in Figure 6. Workload Adjuster is enabled along with RASS, because it is a supplement to RASS. Figure 6 shows that the performance improvements of I/O-intensive workloads are mainly from the LTCR technique; and the performance improvements of computation-intensive workloads are mainly from the RASS technique. When LTCR is enabled, the throughputs of I/O-intensive workloads, HBase and MongoDB, are significantly increased by 41.1% and 44.7%, respectively. However, it barely increases the throughputs of PageRank and Kmeans. Further enabling RASS (with Workload Adjuster) can effectively improve the throughputs of all the workloads.

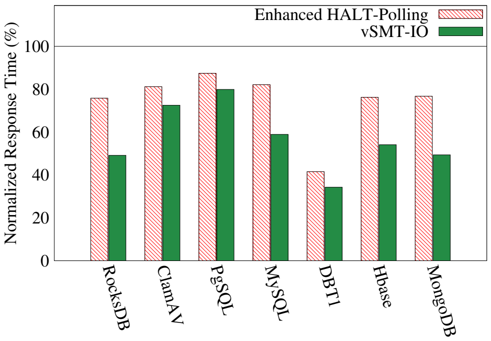


Figure 7: Response times of RocksDB, ClamAV, PgSQL, MySQL, DBT1, HBase, and MongoDB normalized to those with vanilla KVM (shown with the horizontal line at 100%).

Some benchmarks report response times. Figure 7 compares how their response times are reduced with vSMT-IO and HALT-Polling. Relative to vanilla KVM, HALT-Polling reduces the response times by 28.7% on average. vSMT-IO can reduce the response times by larger percentages (50.8% on average). To investigate how vSMT-IO reduces response times, we monitor the state changes of the vCPUs during the executions of these benchmarks, collect the time spent by vCPUs at the following states: 1) ***Running***, including context retention, on a hyperthread, 2) ***Ready*** and waiting to be scheduled, 2) ***Waiting*** for an event. In Table 7, for each benchmark, we show the time (in milliseconds) spent in these states for serving a request.

Benchmark	Vallina KVM			Enhanced HALT-Polling			vSMT-IO		
	Run	Ready	Wait	Run	Ready	Wait	Run	Ready	Wait
RocksDB	116.2	132.6	378.1	131.7	88.0	305.4	129.8	69.0	237.2
ClamAV	15.2	45.7	10.9	12.9	29.7	10.5	11.0	21.1	9.5
PgSQL	14.7	37.6	10.1	12.3	27.1	9.4	13.5	19.7	8.7
MySQL	111.4	319.7	88.9	90.7	167.9	89.5	87.5	136.7	80.6
DBT1	346.2	1831.4	1390.2	361.6	842.9	1035.8	306.9	643.2	641.6
HBase	266.2	655.0	901.8	237.8	323.3	795.9	241.6	315.0	654.3
MongoDB	376.1	528.6	1444.1	365.3	345.2	1276.6	351.7	256.0	897.9

Table 7: Time spent by vCPUs in three states when processing a request with vanilla KVM, enhanced HALT-Polling, and vSMT-IO.

The response times are reduced with vSMT-IO mainly because vCPUs spend less time on waiting to be scheduled or for events. As shown in Table 7, vSMT-IO can significantly reduce the time in the *Ready* state (53.6% on average). This is because context retention reduces context switches between vCPUs, and thus reduces the scheduling delay associated with the switches. We have noticed that the time in the *Waiting* state is substantially reduced for some benchmarks (e.g., DBT1). This is because finising an I/O operation sometimes need the collaboration of multiple vCPUs in the VM. For example, after a vCPU sends out an I/O request and becomes idle, another vCPU may receive the response and must notify the former vCPU by sending it an inter-processor interrupt (IPI). In this case, reducing the *Ready* time of the latter vCPU (i.e., scheduling it earlier) can also reduce the *Waiting* time of the former vCPU.

6.4 Applicability and Overhead

vSMT-IO targets heterogeneous workloads with intensive I/O operations and heavy computation. We want to know how well vSMT-IO performs for the workloads with different heterogeneity. This subsection tests the performance and overhead of vSMT-IO for different workload mixes. We still use 4 VMs to run 4 instances of 2 applications in the experiments. But we change VM sizes (i.e., the number of vCPUs in a VM) to change the workload mix. For example, to make the workload more I/O-intensive, we increase the sizes of the 2 VMs running I/O-intensive benchmarks and reduce the sizes of the VMs running computation-intensive benchmarks. The total number of vCPUs of the 4 VMs is kept fixed (96 vCPUs).

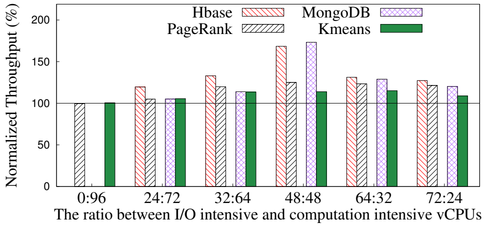


Figure 8: Normalized throughputs of vSMT-IO under different workload mixes. Throughputs are normalized to those with vanilla KVM.

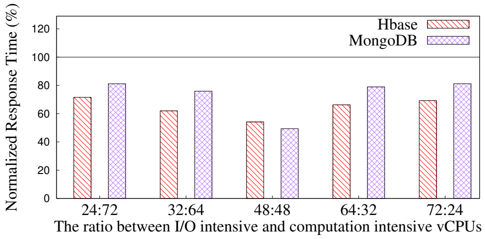


Figure 9: Normalized response times of vSMT-IO under different workload mixes. Response times are normalized to those with vanilla KVM.

Figure 8 shows the normalized throughputs of two benchmark paris, HBase with PageRank, and MongoDB with Kmeans, when the VM sizes for I/O-intensive benchmarks and computation-intensive benchmarks are changed from (12,36) to (36,12). (The ratios of the vCPUs running these benchmarks vary from 24:72 to 72:24.) Figure 9 shows the response times of HBase and MongoDB in these experiments. Though vSMT-IO can improve performance for all these workload mixes, it improves performance by the largest percentages when the number of vCPUs running I/O-intensive benchmarks is the same as the number of vCPUs running computation-intensive vCPUs.

We also run PageRank and Kmeans in two VMs with 48 vCPUs each, and show the normalized throughputs (labeled with "0:96") in Figure 8. Because both benchmarks are computation intensive, there is no space for vSMT-IO to improve performance. The performance difference between vSMT-IO and vanilla KVM is unnoticeable (less than 2%). This shows that the overhead of vSMT-IO is very low.

We have also evaluated the performance of vSMT-IO with 8 VMs (192 vCPUs). We find that vSMT-IO consistently shows better performance than vanilla KVM and enhanced HALT-Polling, for heterogeneous workloads; but the performance improvement is similar to that with 4 VMs. The performance advantage of vSMT-IO is more determined by the mix of workloads than the number of VMs on each server.

7 Related Work

Improving I/O performance in virtualized systems. I/O performance problems in virtualized systems have been intensively studied; and various solutions have been proposed, including shortening time slices [55–58], task-aware priority boosting [17, 18, 59–69], and task consolidation [19, 70–73]. These solutions are not designed for SMT processors, and are orthogonal to our work. Shortening time slices of vCPUs can reduce the latency of I/O workloads in virtualized systems. However, it may incur significant performance degradation caused by context switches. Task-aware priority boosting improves I/O performance in virtualized systems by prioritizing I/O-intensive workloads. For instance, xBalloon [17] maintains the high priority of I/O-intensive workloads by reserving CPU resource for them. However, this may hurt the performance of computation-intensive workloads. vMigrater [19] prioritizes I/O-intensive workloads by migrating them away from to-be-descheduled vCPUs to other vCPUs, such that they can keep running and generating I/O requests. However, it is designed for VMs with multiple vCPUs, and may incur high workload migration cost. Task consolidation solutions can improve I/O performance by reducing the descheduling and rescheduling of vCPUs. They consolidate workloads onto fewer vCPUs if the workloads are I/O-intensive, such that these vCPUs can be kept active with relatively low cost. These solutions may also incur high cost due to frequent workload migrations. Polling is used in these solutions to keep vCPUs active. This is inefficient on SMT processors and can be improved by replacing polling with context retention.

Symbiotic scheduling aims to maximize the throughput of SMT processors by selecting the tasks with complementary resource demands and coscheduling them on the same SMT core [5–10]. For instance, SOS (Sample, Optimize, Symbiosis) and its variants [5–10, 35] sample task executions when they are coscheduled onto the same core, and preferentially coschedule those with small slowdowns. These solutions only target processor throughput, and cannot be used to improve the performance of I/O-intensive workloads.

Other scheduling solutions for SMT processors. Instead of maximizing processor throughput, some scheduling solutions aim to secure resources for individual tasks on SMT processors to ensure their decent performance [11, 35, 74, 75]. For instance, ELFEN [11] aims to ensure the high performance of latency-critical tasks when they are collocated with batch tasks on SMT processors. It puts a latency-critical task and batch tasks on different hardware threads in the same core, and "blocks" batch tasks when the latency-ciritical task is making progress. The efficiency is low with this solution, because each core has only one active hardware thread at any moment, and resource is underutilized. Tasks on the same SMT processor may not share the resources in a fair way. Various solutions have been proposed to enforce fairness among the tasks in a SMT-enabled system [76–78]. For instance, progress-aware scheduler [76] periodically estimates the progress of tasks, and prioritizes the tasks with relatively slow progress. vSMT-IO is orthogonal to these solutions. It increases efficiency to improve both CPU performance and I/O performance.

8 Conclusion and Future Work

Despite the prevalence of SMT processors, the problems with how to improve I/O performance and efficiency on SMT processors are surprisingly under-studied. Existing techniques used in CPU schedulers to improve I/O performance are seriously inefficient on SMT processors, making it difficult to achieve high CPU throughput and high I/O throughput. Leveraging the hardware feature of SMT processors, the paper designs vSMT-IO as an effective solution. The key technique in vSMT-IO is context retention. vSMT-IO targets virtualized clouds and x86 systems and addresses a few challenges in implementing context retention in real systems. Extensive experiments confirm its effectiveness.

NUMA systems have become ubiquitous. Though our evaluation demonstrates that vSMT-IO achieves better performance than competing solutions, the designs in vSMT-IO have not been optimized for NUMA systems. As future work, we want to make vSMT-IO "NUMA-aware" to further improve its performance. For example, the workload adjuster can be enhanced by adjusting workloads within each NUMA node before it migrates vCPUs across NUMA nodes.

9 Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Dr. James Bottomley for his helpful suggestions as the shepherd for this paper. This work is funded in part by the US National Science Foundation under a grant CCF 1617749, HK RGC ECS (No.27200916), HK RGC GRF (No.17207117 and No. 17202318), and a Croucher Innovation Award.

References

[1] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/#instance-details.

[2] SMT Configurations in VMWARE. https://bit.ly/1LxQTiW.

[3] Introducing hyperthreading into azure vms. https://azure.microsoft.com/en-us/blog/introducing-the-new-dv3-and-ev3-vm-sizes/.

[4] Google Cloud Virtual Machine Types. https://cloud.google.com/compute/docs/machine-types.

[5] Allan Snavely and Dean M Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, 35(11):234–244, 2000.

[6] Jun Nakajima and Venkatesh Pallipadi. Enhancements for hyper-threading technology in the operating system: Seeking the optimal scheduling. In *WIESS*, pages 25–38, 2002.

[7] Kefeng Deng, Kaijun Ren, and Junqiang Song. Symbiotic scheduling for virtual machines on SMT processors. In *2012 Second International Conference on Cloud and Green Computing*, pages 145–152. IEEE, 2012.

[8] James R Bulpin and Ian Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*, pages 399–402, 2005.

[9] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. L1-bandwidth aware thread allocation in multicore SMT processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 123–132. IEEE Press, 2013.

[10] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, volume 33, pages 10–17, 2006.

[11] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference*, pages 309–322, 2016.

[12] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip multi processing aware linux kernel scheduler. In *Linux Symposium*, page 193. Citeseer, 2005.

[13] Matt Liebowitz, Christopher Kusek, and Rynardt Spies. *VMware VSphere performance: designing CPU, memory, storage, and networking for performance-intensive workloads*. John Wiley & Sons, 2014.

[14] KVM: Dynamic Halt Polling Patches. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=aca6ff29c4063a8d467cdee241e6b3bf7dc4a171.

[15] Dynamic Halt Polling Technique. https://lkml.org/lkml/2017/6/22/296.

[16] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[17] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 269–281. ACM, 2017.

[18] Luwei Cheng and Cho-Li Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 2. ACM, 2012.

[19] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. Effectively mitigating i/o inactivity in vcpu scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[20] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, 2013. USENIX.

[21] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[22] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[23] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58. IEEE, 1999.

[24] Intel 64 and ia-32 architectures developer's manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[25] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Winter USENIX Conference*, page 405–420, 1993.

[26] Damien Le Moal. I/o latency optimization with polling. *Vault–Linux Storage and Filesystem*, 2017.

[27] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[28] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *FAST*, volume 12, pages 15–15, 2012.

[29] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[30] L1 Terminal Fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[31] Flushing L1 Data Cache When a vCPU Enters the Guest OS. https://lore.kernel.org/patchwork/patch/974356/.

[32] Flushing TLB When a vCPU Enters the Guest OS. https://lwn.net/Articles/740363/.

[33] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *ACM SIGPLAN Notices*, volume 48, pages 369–380. ACM, 2013.

[34] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using SMT to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 750–761. ACM, 2019.

[35] Allan Snavely, Dean M Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 66–76. ACM, 2002.

[36] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A performance counter architecture for computing accurate cpi components. *ACM SIGOPS Operating Systems Review*, 40(5):175–184, 2006.

[37] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in SMT processors. *ACM Sigplan Notices*, 44(3):133–144, 2009.

[38] Josue Feliu, Stijn Eyerman, Julio Sahuquillo, and Salvador Petit. Symbiotic job scheduling on the ibm power8. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 669–680. IEEE, 2016.

[39] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 91–102. ACM, 2010.

[40] The HALT-Polling Kernel Module. `https://patchwork.kernel.org/patch/11030651/`, 2019.

[41] The Halt Polling Technique. `https://lwn.net/Articles/384146/`.

[42] Redis In-memory Key-Value Database. `http://redis.io/`.

[43] Apache Hadoop Systems. `http://hadoop.apache.org/core/`.

[44] Yahoo! Cloud Serving Benchmark. `https://github.com/brianfrankcooper/YCSB`, 2004.

[45] MySQL Database. `http://www.mysql.com/`, 2014.

[46] Apache Web Server. `http://www.apache.org`, 2012.

[47] Clam AntiVirus Benchmarks. `http://www.clamav.net/`.

[48] RocksDB NoSQL Storage System. `https://rocksdb.org/`.

[49] PostgreSQL DBMS Benchmarks. `https://www.postgresql.org`, 2012.

[50] Apache spark benchmarks. `https://spark.apache.org/examples.html`.

[51] TPC-W Database Benchmarks. `http://osdldbt.sourceforge.net/`.

[52] XGBOOST Runtime System. `http://dmlc.cs.washington.edu/xgboost.html`.

[53] Sockperf. `https://github.com/Mellanox/sockperf`.

[54] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[55] Cong Xu, Sahan Gamage, Pawan N Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[56] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 394–405. IEEE Computer Society, 2014.

[57] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Accelerating critical os services in virtualized systems with flexible micro-sliced cores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 29:1–29:14, New York, NY, USA, 2018. ACM.

[58] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 3:1–3:14, 2016.

[59] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[60] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[61] Xiaoning Ding, Phillip B Gibbons, and Michael A Kozuch. A hidden cost of virtualization when scaling multicore applications. In *HotCloud*, 2013.

[62] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[63] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vread: Efficient data access for hadoop in virtualized clouds. In *Proceedings of the 16th Annual Middleware Conference*, pages 125–136. ACM, 2015.

[64] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[65] Hui Lu, Cong Xu, Cheng Cheng, Ramana Kompella, and Dongyan Xu. vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 453–460. IEEE, 2015.

[66] Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Opportunistic flooding to improve tcp transmit performance in virtualized clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 24. ACM, 2011.

[67] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138. ACM, 2015.

[68] Ron C Chiang and H Howie Huang. Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.

[69] Weiwei Jia Jianchen Shan and Xiaoning Ding. Rethinking the scalability of multicore applications on big virtual machines. In *IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2017.

[70] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not VCPUs. In *APSys 2013*, pages 1:1–1:7, 2013.

[71] Luwei Cheng, Jia Rao, and Francis Lau. vScale: automatic and efficient processor scaling for smp virtual machines. In *EuroSys 2016*, page 2. ACM, 2016.

[72] Xiaoning Ding, Phillip B Gibbons, Michael A Kozuch, and Jianchen Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, 2014.

[73] Xiang Song, Haibo Chen, Binyu Zang, X SONG, H CHEN, and B ZANG. Characterizing the performance and scalability of many-core applications on virtualized platforms. *Parallel Processing Institute Technical Report Number: FDUPPITR-2010*, 2, 2010.

[74] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE Computer Society, 2014.

[75] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–27. IEEE, 2019.

[76] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Addressing fairness in SMT multicores with a progress-aware scheduler. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 187–196. IEEE, 2015.

[77] David Koufaty and Deborah T Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[78] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2012.

Lightweight Preemptible Functions

Sol Boucher,[*] Anuj Kalia,[†] David G. Andersen,[*] and Michael Kaminsky[‡*]
[*]*Carnegie Mellon University* [†]*Microsoft Research* [‡]*BrdgAI*

Abstract

Lamenting the lack of a natural userland abstraction for preemptive interruption and asynchronous cancellation, we propose lightweight preemptible functions, a mechanism for synchronously performing a function call with a precise timeout that is lightweight, efficient, and composable, all while being portable between programming languages. We present the design of *libinger*, a library that provides this abstraction, on top of which we build *libturquoise*, arguably the first general-purpose and backwards-compatible preemptive thread library implemented entirely in userland. Finally, we demonstrate this software stack's applicability to and performance on the problems of combatting head-of-line blocking and time-based DoS attacks.

1 Introduction

After years of struggling to gain adoption, the coroutine has finally become a mainstream abstraction for cooperatively scheduling function invocations. Languages as diverse as C#, JavaScript, Kotlin, Python, and Rust now support "async functions," each of which expresses its dependencies by "awaiting" a **future** (or promise); rather than polling, the language yields if the awaited result is not yet available.

Key to the popularity of this concurrency abstraction is the ease and seamlessness of parallelizing it. Underlying most futures runtimes is some form of green threading library, typically consisting of a scheduler that distributes work to a pool of OS-managed worker threads. Without uncommon kernel support (e.g., scheduler activations [3]), however, this logical threading model renders the operating system unaware of individual tasks, meaning context switches are purely cooperative. This limitation is common among userland thread libraries, and illustrates the need for a mechanism for *preemptive* scheduling at finer granularity than the kernel thread.

In this paper, we propose an abstraction for calling a function with a timeout: Once invoked, the function runs on the same thread as the caller. Should the function time out, it is preempted and its execution state is returned as a continuation in case the caller later wishes to resume it. The abstraction is exposed via a wrapper function reminiscent of a thread spawn interface such as `pthread_create()` (except *synchronous*). Despite their synchronous nature, **preemptible functions** are useful to programs that are parallel or rely on asynchronous I/O; indeed, we later demonstrate how our abstraction composes with futures and threads.

The central challenge of introducing preemption into the contemporary programming model is supporting existing code. Despite decades of improvement focused on thread safety, modern systems stacks still contain critical nonreentrancy, ranging from devices to the dynamic memory allocator's heap region. Under POSIX, code that interrupts other user code is safe only if it restricts itself to calling async-signal-safe (roughly, reentrant) functions [27]. This restriction is all too familiar to those programmers who have written signal handlers: it is what makes it notoriously difficult to write nontrivial ones. Preemption of a timed function constitutes its interruption by the rest of the program. This implies that *the rest of the program* should be restricted to calling reentrant functions; needless to say, such a rule would be crippling. Addressing this problem is one of the main contributions of this paper. Our main insight here, as shown in Figure 1, is that some libraries are naturally reentrant, while many others can be made reentrant by automatically cloning their internal state so that preempting one invocation does not leave the library "broken" for concurrent callers.

The most obvious approach to implementing preemptible functions is to map them to OS threads, where the function would run on a new thread that could be cancelled upon timeout. Unfortunately, cancelling a thread is also hard. UNIX's pthreads provide asynchronous cancelability, but according to the Linux documentation, it

> is rarely useful. Since the thread could be cancelled at *any* time, it cannot safely reserve resources (e.g., allocating memory with `malloc()`), acquire mutexes, semaphores, or locks, and so on... some internal data structures (e.g., the linked list of free blocks managed by the `malloc()` family of functions) may be left in an inconsistent state if cancellation occurs in the middle of the function call [25].

[†] This author was at Carnegie Mellon during this project.
[‡] This author was *not* at Carnegie Mellon during this project.

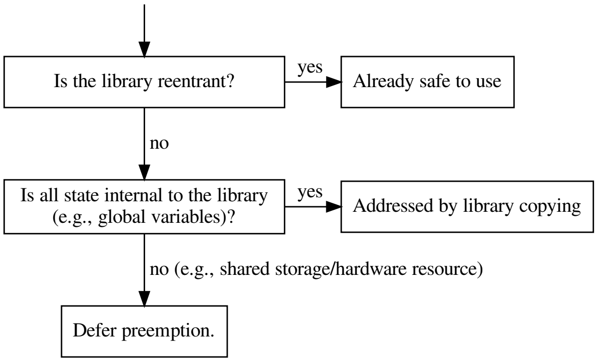

Is the library reentrant?
yes
Already safe to use
no
Is all state internal to the library (e.g., global variables)?
yes
Addressed by library copying
no (e.g., shared storage/hardware resource)
Defer preemption.

Figure 1: Taxonomy of support for library code. In practice, we always apply one of the two mitigations. Library copying is used by default, and is discussed in Sections 5.1 and 5.2. Deferred preemption is needed to preserve the semantics of `malloc()` and users of uncopyable resources such as file descriptors or network adapters, and is applied according to a whitelist, as described in Section 5.5.

The same is true on Windows, whose API documentation warns that asynchronously terminating a thread

> can result in the following problems: If the target thread owns a critical section, the critical section will not be released. If the target thread is allocating memory from the heap, the heap lock will not be released...

and goes on from there [28].

One might instead seek to implement preemptible functions via the UNIX `fork()` call. Assuming a satisfactory solution to the performance penalty of this approach, one significant challenge would be providing bidirectional object visibility and ownership. In a model where each timed function executes in its own child process, not only must data allocated by the parent be accessible to the child, but the opposite must be true as well. The fact that the child may terminate before the parent raises allocation lifetime questions. And all this is without addressing the difficulty of even calling `fork()` in a multithreaded program: because doing so effectively cancels all threads in the child process except the calling one, the child process can experience the same problems that plague thread cancellation [4].

These naïve designs share another shortcoming: in reducing preemptible functions to a problem of parallelism, they hurt performance by placing thread creation on the critical path. Thus, the state-of-the-art abstractions' high costs limit their composability. We observe that, when calling a function with a timeout, it is concurrency alone—and not parallelism—that is fundamental. Leveraging this key insight, we present a design that *separates interruption from asynchrony* in order to provide *preemption at granularities in the tens of microseconds*, orders of magnitude finer than contemporary OS schedulers' millisecond timescales. Our research prototype[1] is implemented entirely in userland, and requires neither custom compiler or runtime support nor managed runtime features such as garbage collection.

This paper makes three primary contributions: (1) It proposes function calls that return a continuation upon preemption, a novel primitive for unmanaged languages. (2) It introduces selective relinking, a compiler-agnostic approach to automatically lifting safety restrictions related to nonreentrancy. (3) It demonstrates how to support asynchronous function cancellation, a feature missing from state-of-the-art approaches to preemption, even those that operate at the coarser granularity of a kernel thread.

2 Related work

A number of past projects (Table 1) have sought to provide bounded-time execution of chunks of code at subprocess granularity. For the purpose of our discussion, we refer to a portion of the program whose execution should be bounded as **timed code** (a generalization of a preemptible function); exactly how such code is delineated depends on the system's interface.

Interface notwithstanding, the systems' most distinguishing characteristic is the mechanism by which they enforce execution bounds. At one end of the spectrum are **cooperative** multitasking systems where timed code voluntarily cedes the CPU to another task via a runtime check. (This is often done implicitly; a simple example is a compiler that injects a conditional branch at the beginning of any function call from timed code.) Occupying the other extreme are **preemptive** systems that externally pause timed code and transfer control to a scheduler routine (e.g., via an interrupt service routine or signal handler, possibly within the language's VM).

The cooperative approach tends to be unable to interrupt two classes of timed code: (1) **blocking-call** code sections that cause long-running kernel traps (e.g., by making I/O system calls), thereby preventing the interruption logic from being run; and (2) **excessively-tight loops** whose body does not contain any yield points (e.g., spin locks or long-running CPU instructions). Although some cooperative systems refine their approach with mechanisms to tolerate either blocking-call code sections [1] or excessively-tight loops [32], we are not aware of any that are capable of handling both cases.

One early instance of timed code support was the *engines* feature of the Scheme 84 language [15]. Its in-

[1] Our system is open source; the code is available from `efficient.github.io/#lpf`.

			Dependencies		Third-party code support	
System	Preemptive	Synchronous	In userland	Works without GC	Preemptible	Works without recompiling
Scheme engines	✓*	✓	✓		†	✓
Lilt		✓	✓		†*	—
goroutines			✓		†*	—
C∀	✓		✓	✓	†*	—
RT library	✓		✓	✓		✓
Shinjuku	✓			✓	†	
libinger	✓	✓	✓	✓	✓	✓

✓* = the language specification leaves the interaction with blocking system calls unclear
† = assuming the third-party library is written in a purely functional (stateless) fashion
†* = the third-party code must be written in the language without foreign dependencies
(beyond simple recompilation, this necessitates porting)

Table 1: Systems providing timed code at sub-process granularity

terface was a new `engine` keyword that behaved similarly to `lambda`, but created a special "thunk" accepting as an argument the number of ticks (abstract time units) it should run for. The caller also supplied a callback function to receive the timed code's return value upon successful completion. Like the rest of the Scheme language, engines were stateless: whenever one ran out of computation time, it would return a replacement engine recording the point of interruption. Engines' implementation relied heavily on Scheme's managed runtime, with ticks corresponding to virtual machine instructions and cleanup handled by the garbage collector. Although the paper mentions timer interrupts as an alternative, it does not evaluate such an approach.

Lilt [32] introduced a language for writing programs with statically-enforced timing policies. Its compiler tracks the possible duration of each path through a program and inserts yield operations wherever a timeout could possibly occur. Although this approach requires assigning the execution limit at compile time, the compiler is able to handle excessively-tight loops by instrumenting backward jumps. Blocking-call functions remained a challenge, however: handling them would have required operating system support, reminiscent of *Singularity*'s static language-based isolation [12].

Some recent languages offer explicit userland threading, which could be used to support timed code. One example is the Go language's [1] *goroutines*. Its runtime includes a cooperative scheduler that conditionally yields at function call sites. This causes problems with tight loops, which require the programmer to manually add calls to the `runtime.Gosched()` yield function [7].

The solutions described thus far all assume languages with a heavyweight, garbage-collected runtime. However, two recent systems seek to support timed code with fewer dependencies: the *C∀* language [8] and a C thread library for realtime systems (here, "*RT*") developed by Mollison and Anderson [22]. Both perform preemption using timer interrupts, as proposed in the early Scheme engines literature. They install a periodic signal handler for scheduling tasks and migrating them between cores, a lightweight approach that achieves competitive scheduling latencies. However, as explained later in this section, the compromise is interoperability with existing code.

Shinjuku [17] is an operating system designed to perform preemption at microsecond scale. Built on the Dune framework [5], it runs tasks on a worker thread pool controlled by a single centralized dispatcher thread. The latter polices how long each task has been running and sends an inter-processor interrupt (IPI) to any worker whose task has timed out. The authors study the cost of IPIs and the overheads imposed by performing them within a VT-x virtual machine, as required by Dune. They then implement optimizations to reduce these overheads at the expense of Shinjuku's isolation from the rest of the system.

As seen in Section 1, nonreentrant interfaces are incompatible with externally-imposed time limits. Because such interfaces are prolific in popular dependencies, no prior work allows timed code to transparently call into third-party libraries. Scheme engines and Lilt avoid this issue by only supporting functional code, which cannot have shared state. Go is able to preempt goroutines written in the language itself, but a goroutine that makes any foreign calls to other languages is treated as nonpreemptible by the runtime's scheduler [11]. The C∀ language's preemption model is only safe for functions guarded by its novel monitors: the authors caution that "any challenges that are not [a result of extending monitor semantics] are considered as solved problems and therefore not discussed." With its focus on realtime embedded systems, RT assumes that the timed code in its threads will avoid shared state; this assumption

```
struct linger_t {
    bool is_complete;
    cont_t continuation;
};

linger_t launch(Function func,
                u64 time_us,
                void *args);
void resume(linger_t *cont, u64 time_us);
```

Listing 1: Preemptible functions core interface

```
linger = launch(task, TIMEOUT, NULL);
if (!linger.is_complete) {
    // Save @linger to a task queue to
    // resume later
    task_queue.push(linger);
}

// Handle other tasks
...
// Resume @task at some later point
linger = task_queue.pop();
resume(&linger, TIMEOUT);
```

Listing 2: Preemptible function usage example

mostly precludes calls to third-party libraries, though the system supports the dynamic memory allocator by treating it as specifically nonpreemptible. Rather than dealing with shared state itself, Shinjuku asks application authors to annotate any code with potential concurrency concerns using a nonpreemptible `call_safe()` wrapper.

3 Timed functions: *libinger*

To address the literature's shortcomings, we have developed *libinger*,[2] a library providing a small API for timed function dispatch (Listing 1):

- `launch()` invokes an ordinary function `func` with a time cap of `time_us`. The call to `launch()` returns when `func` completes, or after approximately `time_us` microseconds if `func` has not returned by then. In the latter case, *libinger* returns an opaque continuation object recording the execution state.
- `resume()` causes a preemptible function to continue after a timeout. If execution again times out, `resume()` updates its continuation so the process may be repeated. Resuming a function that has already returned has no effect.

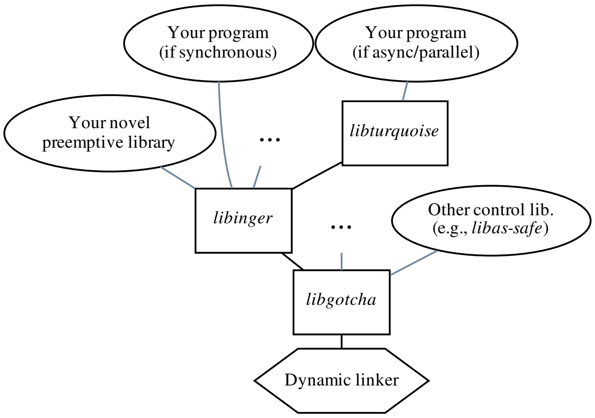


Figure 2: Preemptible functions software stack. Hexagonal boxes show the required runtime environment. Rectangular boxes represent components implementing the preemptible functions abstraction. Ovals represent components built on top of these. A preemptible function's body (i.e., `func`) may be defined directly in your program, or in some other loaded library.

Listing 2 shows an example use of *libinger* in a task queue manager designed to prevent latency-critical tasks from blocking behind longer-running ones. The caller invokes a task with a timeout. If the task does not complete within the allotted time, the caller saves its continuation in the task queue, handles other tasks, and later resumes the first task.

In accordance with our goal of language agnosticism, *libinger* exposes both C and Rust [2] APIs. To demonstrate the flexibility and composability of the preemptible function abstraction, we have also created *libturquoise*, a preemptive userland thread library for Rust, by porting an existing futures-based thread pool to *libinger*. We discuss this system in Section 4.

Figure 2 shows a dependency graph of the software components comprising the preemptible functions stack. The *libinger* library itself is implemented in approximately 2,500 lines of Rust. To support calls to nonreentrant functions, it depends on another library, *libgotcha*, which consists of another 3,000 lines of C, Rust, and x86-64 assembly. We cover the details in Section 5.

We now examine *libinger*, starting with shared state.

3.1 Automatic handling of shared state

As we found in Section 1, a key design challenge facing *libinger* is the shared state problem: Suppose a preemptible function F calls a stateful routine in a third-party library L, and that F times out and is preempted by *libinger*. Later, the user invokes another timed function F_0, which also calls a stateful routine in L. This pattern involves an unsynchronized concurrent access to L. To avoid introducing such bugs, *libinger* must hide state modifications in L by F from the execution of F_0.

One non-solution to this problem is to follow the ap-

[2] In the style of GNU's *libiberty*, we named our system for the command-line switch used to link against it. As the proverb goes, "Don't want your function calls to linger? Link with `-linger`."

proach taken by POSIX signal handlers and specify that preemptible functions may not call third-party code, but doing so would severely limit their usefulness (Section 2). We opt instead to automatically and dynamically create copies of L to isolate state from different timed functions. Making this approach work on top of existing systems software required solving many design and implementation challenges, which we cover when we introduce *libgotcha* in Section 5.

3.2 Safe concurrency

Automatically handling shared state arising from non-reentrant library interfaces is needed because the sharing is transparent to the programmer. A different problem arises when a programmer explicitly shares state between a preemptible function and any other part of the program. Unlike third-party library authors, this programmer knows they are using preemptible functions, a concurrency mechanism.

When using the C interface, the programmer bears complete responsibility for writing race-free code (e.g., by using atomics and mutexes wherever necessary). The *libinger* Rust API, however, leverages the language's first-class concurrency support to prevent such mistakes from compiling: `launch()`'s signature requires the wrapped function to be `Send` safe (only reference state in a thread-safe manner) [29].

While the Rust compiler rejects all code that shares state unsafely, it is still possible to introduce correctness bugs such as deadlock [30]. For example, a program might block on a mutex held by the preemptible function's caller (recall that invocation is synchronous, so blocking in a preemptible function does not cause it to yield!). It is sometimes necessary to acquire such a mutex, so *libinger* provides a way to do it: The API has an additional function, `pause()`, that is a rough analog of yield. After performing a try-lock operation, a preemptible function can call `pause()` to immediately return to its caller as if it had timed out. The caller can tell whether a function paused via a flag on its continuation.

3.3 Execution stacks

When a preemptible function times out, *libinger* returns a continuation object. The caller might pass this object around the program, which could later `resume()` from a different stack frame. To handle this case, the `launch()` function switches to a new, dedicated stack just before invoking the user-provided function. This stack is then stored in the continuation alongside the register context.

Because of the infeasibility of moving these stacks after a function has started executing, *libinger* currently heap-allocates large 2-MB stacks so it can treat them as having fixed size. To avoid an order of magnitude slowdown from having such large dynamic allocations on the critical path, *libinger* preallocates a pool of reusable stacks when it is first used.

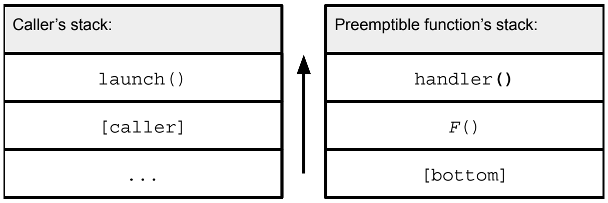


Figure 3: The stacks just before a timeout. Upon discovering that the preemptible function has exceeded its time bound, the handler jumps into the `launch()` (or `resume()`) function, which in turn returns to the original call site, removing its own stack frame in the process.

3.4 Timer interrupts

Whenever *libinger* is executing a user-provided function, we enable fine-grained timer interrupts to monitor that function's elapsed running time. A timer interrupt fires periodically,[3] causing our signal handler to be invoked. If the function exceeds its timeout, this handler saves a continuation by dumping the machine's registers. It then performs an unstructured jump out of the signal handler and back into the `launch()` or `resume()` function, switching back to the caller's stack as it does so. Figure 3 shows the two stacks of execution that are present while the signal handler is running.

A subsequent `resume()` call restores the registers from the stored continuation, thereby jumping back into the signal handler. The handler returns, resuming the preemptible function from the instruction that was executing when the preemption signal arrived.

To support blocking system calls, we use the `SA_RESTART` flag when installing the signal handler to instruct libc to restart system calls that are interrupted by the signal [26]. We direct signals at the process's specific threads that are running preemptible functions by allocating signal numbers from a pool, an approach that limits the number of simultaneous invocations to the number of available signals; this restriction could be lifted by instead using the Linux-specific `SIGEV_THREAD_ID` timer notification feature [31].

3.5 Cancellation

Should a caller decide not to finish running a timed-out preemptible function, it must deallocate it. In Rust, deal-

[3] Signal arrival is accurate to microsecond timescales, but exhibits a warmup effect. For simplicity, we use a fixed signal frequency for all preemptible functions, but this is not fundamental to the design. In the future, we plan to adjust each function's frequency based on its timeout, and to delay the first signal until shortly before the prescribed timeout (in the case of longer-running functions).

location happens implicitly via the `linger_t` type's destructor, whereas users of the C interface are responsible for explicitly calling the *libinger* `cancel()` function.

Cancellation cleans up *libinger* resources allocated by `launch()`; however, the current implementation does not automatically release resources already claimed by the preemptible function itself. While the lack of a standard resource deallocation API makes such cleanup inherently hard to do in C, it is possible in Rust and other languages in which destructor calls are ostensibly guaranteed. For instance, the approach proposed by Boucher et al. [6] could be employed to raise a panic (exception) on the preemptible function's stack. This in turn would cause the language runtime to unwind each stack frame, invoking local variables' destructors in the process.

4 Thread library: *libturquoise*

Until now, we have limited our discussion to synchronous, single-threaded programs. In this section, we will show that the preemptible function abstraction is equally relevant to asynchronous and parallel programs, and that it composes naturally with both futures and threads. As a proof of concept, we have created *libturquoise*,[4] a preemptive userland thread library.

That *libturquoise* provides preemptive scheduling is a significant achievement: *Shinjuku* observes that "there have been several efforts to implement efficient, user-space thread libraries. They all focus on cooperative scheduling" [17]. (Though *RT* from Section 2 could be a counterexample, its lack of nonreentrancy support renders it far from general purpose.) We attribute the dearth of preemptive userland thread libraries to a lack of natural abstractions to support them.

Before presenting the *libturquoise* design, we begin with some context about futures.

4.1 Futures and asynchronous I/O

As mentioned in Section 1, futures are a primitive for expressing asynchronous program tasks in a format amenable to cooperative scheduling. Structuring a program around futures makes it easy to achieve low latency by enabling the runtime to reschedule slow operations off the critical path. Alas, blocking system calls (which cannot be rescheduled by userland) defeat this approach.

The community has done extensive prior work to support asynchronous I/O via result callbacks [19, 18, 20, 23]. Futures runtimes such as Rust's Hyper [16] have adapted this approach by providing I/O libraries whose functions return futures. Rather than duplicate this work, we have integrated preemptible functions with futures so they can leverage it.

```
function PreemptibleFuture(Future fut,
                           Num timeout):
    function adapt():
        // Poll wrapped future in the usual way
        while poll(fut) == NotReady:
            pause()
    fut.linger = launch(adapt, CREATE_ONLY)
    fut.timeout = timeout
    return fut

// Custom polling logic for preemptible futures
function poll(PreemptibleFuture fut):
    resume(fut.linger, fut.timeout);
    if has_finished(fut.linger):
        return Ready
    else
        if called_pause(fut.linger):
            notify_unblocked(fut.subscribers)
        return NotReady
```

Listing 3: Futures adapter type (pseudocode)

4.2 Preemptible futures

For seamless interoperation between preemptible functions and the futures ecosystem, we built a preemptible future adapter that wraps the *libinger* API. Like a normal future, a preemptible future yields when its result is not ready, but it can also time out.

Each language has its own futures interface, so preemptible futures are not language agnostic like the preemptible functions API. Fortunately, they are easy to implement by using `pause()` to propagate cooperative yields across the preemptive function boundary. We give the type construction and polling algorithm in Listing 3; our Rust implementation is only 70 lines.

4.3 Preemptive userland threading

We built the *libturquoise* thread library by modifying the *tokio-threadpool* [13] work-stealing scheduler from the Rust futures ecosystem. Starting from version 0.1.16 of the upstream project, we added 50 lines of code that wrap each incoming task in a preemptible future.

Currently, *libturquoise* assigns each future it launches or resumes the same fixed time budget, although this design could be extended to support multiple job priorities. When a task times out, the scheduler pops it from its worker thread's job queue and pushes it to the incoming queue, offering it to any available worker for rescheduling after all other waiting jobs have had a turn.

5 Shared state: *libgotcha*

We now present one more artifact, *libgotcha*. Despite the name, it is more like a runtime that isolates hidden shared state within an application. Although the rest of the program does not interact directly with *libgotcha*, its

[4] so called because it implements "green threading with a twist"

```c
static bool two;
bool three;

linger_t caller(const char *s, u64 timeout) {
    stdout = NULL;
    two = true;
    three = true;
    return launch(timed, timeout, s);
}

void timed(void *s) {
    assert(stdout); // (1)
    assert(two); // (2)
    assert(three); // (3)
}
```

Listing 4: Demo of isolated (1) **vs. shared** (2&3) **state**

presence has a global effect: once loaded into the process image, it employs a technique we call **selective relinking** to dynamically intercept and reroute many of the program's function calls and global variable accesses.

The goal of *libgotcha* is to establish around every preemptible function a memory isolation boundary encompassing whatever third-party libraries that function interacts with (Section 3.1). The result is that the only state shared across the boundary is that explicitly passed via arguments, return value, or closure—the same state the application programmer is responsible for protecting from concurrency violations (Section 3.2). Listing 4 shows the impact on an example program, and Figure 1 classifies libraries by how *libgotcha* supports them.

Note that *libgotcha* operates at runtime; this constrains its visibility into the program, and therefore the granularity of its operation, to shared libraries. It therefore assumes that the programmer will dynamically link all third-party libraries, since otherwise there is no way to tell them apart from the rest of the program at runtime. We feel this restriction is reasonable because a programmer wishing to use *libinger* or *libgotcha* must already have control over their project's build in order to add the dependency.

Before introducing the *libgotcha* API and explaining selective relinking, we now briefly motivate the need for *libgotcha* by demonstrating how existing system interfaces fail to provide the required type of isolation.

5.1 Library copying: namespaces

Expanding a preemptible function's isolation boundary to include libraries requires providing it with private copies of those libraries. POSIX has long provided a `dlopen()` interface to the dynamic linker for loading shared objects at runtime; however, opening an already-loaded library just increments a reference count, and this function is therefore of no use for making copies.

```c
typedef long libset_t;

bool libset_thread_set_next(libset_t);
libset_t libset_thread_get_next(void);
bool libset_reinit(libset_t);
```

Listing 5: *libgotcha* C interface

Fortunately, the GNU dynamic linker (`ld-linux.so`) also supports Solaris-style **namespaces**, or isolated sets of loaded libraries. For each namespace, `ld-linux.so` maintains a separate set of loaded libraries whose dependency graph and reference counts are tracked independently from the rest of the program [9].

It may seem like namespaces provide the isolation we need: whenever we `launch(`*F*`)`, we can initialize a namespace with a copy of the whole application and transfer control into that namespace's copy of *F*, rather than the original. The problem with this approach is that it breaks the lexical scoping of static variables. For example, Listing 4 would fail assertion (2).

5.2 Library copying: libsets

We just saw that namespaces provide too much isolation for our needs: because of their completely independent dependency graphs, they never encounter any state from another namespace, even according to normal scoping rules. However, we can use namespaces to build the abstraction we need, which we term a **libset**. A libset is like a namespace, except that the program can decide whether symbols referenced within a libset resolve to the same libset or a different one. Control libraries such as *libinger* configure such **libset switches** via *libgotcha*'s private control API, shown in Listing 5.

This abstraction serves our needs: when a `launch(`*F*`)` happens, *libinger* assigns an available `libset_t` exclusively to that preemptible function. Just before calling *F*, it informs *libgotcha* by calling `libset_thread_set_next()` to set the thread's **next libset**: any dynamic symbols used by the preemptible function will resolve to this libset. The thread's **current libset** remains unchanged, however, so the preemptible function itself executes from the same libset as its caller and the two share access to the same global variables.

One scoping issue remains, though. Because dynamic symbols can resolve back to a definition in the same executable or shared object that used them, Listing 5 would fail assertion (3) under the described rules. We want global variables defined in *F*'s object file to have the same scoping semantics regardless of whether they are declared `static`, so *libgotcha* only performs a namespace switch when the use of a dynamic symbol occurs in a different executable or shared library than that symbol's definition.

5.3 Managing libsets

At program start, *libgotcha* initializes a pool of libsets, each with a full complement of the program's loaded object files. Throughout the program's run *libinger* tracks the libset assigned to each preemptible function that has started running but not yet reached successful completion. When a preemptible function completes, *libinger* assumes it has not corrupted its libset and returns it to the pool of available ones. However, if a preemptible function is canceled rather than being allowed to return, *libinger* must assume that its libset's shared state could be corrupted. It unloads and reloads all objects in such a libset by calling `libset_reinit()`.

While *libinger* in principle runs on top of an unmodified `ld-linux.so`, in practice initializing more than one namespace tends to exhaust the statically-allocated thread-local storage area. As a workaround, we build glibc with an increased `TLS_STATIC_SURPLUS`. It is useful to also raise the maximum number of namespaces by increasing `DL_NNS`.

5.4 Selective relinking

Most of the complexity of *libgotcha* lies in the implementation of selective relinking, the mechanism underlying libset switches.

Whenever a program uses a dynamic symbol, it looks up its address in a data structure called the global offset table (GOT). As it loads the program, `ld-linux.so` eagerly resolves the addresses of all global variables and some functions and stores them in the GOT.

Selective relinking works by shadowing the GOT.[5] As soon as `ld-linux.so` finishes populating the GOT, *libgotcha* replaces every entry that should trigger a libset switch with a fake address, storing the original one in its shadow GOT, which is organized by the libset that houses the definition. The fake address used depends upon the type of symbol:

Functions' addresses are replaced by the address of a special function, `procedure_linkage_override()`. Whenever the program tries to call one of the affected functions, this intermediary checks the thread's next libset, looks up the address of the appropriate definition in the shadow GOT, and jumps to it. Because `procedure_linkage_override()` runs between the caller's `call` instruction and the real function, it is written in assembly to avoid clobbering registers. Instead of being linked to their symbol definitions at load time, some function calls resolve lazily the first time they are called: their GOT entries initially point to a special lookup function in the dynamic linker that rewrites the GOT entry when invoked. Such memoization would remove our intermediary, so we alter the ELF relocation entries of affected symbols to trick the dynamic linker into updating our shadow GOT instead.

Global variables' addresses are replaced with a unique address within a mapped but inaccessible page. When the program tries to read or write such an address, a segmentation fault occurs; *libgotcha* handles the fault, disassembles the faulting instruction to determine the base address register of its address calculation,[6] loads the address from this register, computes the location of the shadow GOT entry based on the fake address, checks the thread's next libset, and replaces the register's contents with the appropriate resolved address. It then returns, causing the faulting instruction to be reexecuted with the valid address this time.[7]

5.5 Uninterruptible code: uncopyable

The library-copying approach to memory isolation works for the common case, and allows us to handle most third-party libraries with no configuration. However, in rare cases it is not appropriate. The main example is the `malloc()` family of functions: in Section 1, we observed that not sharing a common heap complicates ownership transfer of objects allocated from inside a preemptible function. To support dynamic memory allocation and a few other special cases, *libgotcha* has an internal whitelist of **uncopyable** symbols.

From *libgotcha*'s perspective, uncopyable symbols differ only in what happens on a libset switch. If code executing in any libset other than the application's **starting libset** calls an uncopyable symbol, a libset switch still occurs, but it returns to the starting libset instead of the next libset; thus, all calls to an uncopyable symbol are routed to a single, globally-shared definition. When the function call that caused one of these special libset switches returns, the next libset is restored to its prior value. The *libgotcha* control API provides one more function, `libset_register_interruptible_callback()`, that allows others to request a notification when one of these libset restorations occurs.

Because it is never safe to preempt while executing in the starting libset, the first thing the *libinger* preemption handler described in Section 3.4 does is check whether the thread's next libset is set to the starting one; if so, it disables preemption interrupts and immediately returns. However, *libinger* registers an interruptible call-

[5] Hence the name *lib**got**cha*.

[6] Although it is possible to generate code sequences that are incompatible with this approach (e.g., because they perform in-place pointer arithmetic on a register rather than using displacement-mode addressing with a base address), we employ a few heuristics based on the context of the instruction and fault; in our experience, these cover the common cases.

[7] This does not break applications with existing segfault handlers: we intercept their calls to `sigaction()`, and forward the signal along to their handler when we are unable to resolve an address ourselves.

back that it uses to reenable preemption as soon as any uncopyable function returns.

5.6 Limitations

The current version of *libgotcha* includes partial support for thread-local storage (TLS). Like other globals, TLS variables are copied along with the libset; this behavior is correct because a thread might call into the same library from multiple preemptible functions. However, we do not yet support migrating TLS variables between threads along with their preemptible function. This restriction is not fundamental: the TLS models we support (general dynamic and local dynamic) use a support function called `__tls_get_addr()` to resolve addresses [10], and *libgotcha* could substitute its own implementation that remapped the running thread's TLS accesses to that of the preemptible function's initial thread when executing outside the starting libset.

While selective relinking supports the ordinary `GLOB_DAT` (eager) and `JUMP_SLOT` (lazy) ELF dynamic relocation types, it is incompatible with the optimized `COPY` class of dynamic variable relocations. The `COPY` model works by allocating space for all libraries' globals in the executable, enabling static linking from the program's code (but not its dynamic libraries'). This transformation defeats selective relinking for two reasons: the use of static linking prevents identifying symbol uses in the executable, and the cross-module migration causes breakages such as failing assertion (3) from Listing 4. When building a program that depends on *libgotcha*, programmers must instruct their compiler to disable `COPY` relocations, as with the `-fpic` switch to GCC and Clang. If *libgotcha* encounters any `COPY` relocations in the executable, it prints a load-time warning.

Forsaking `COPY` relocations does incur a small performance penalty, but exported global variables are rare now that thread safety is a pervasive concern in system design. Even the POSIX-specified `errno` global is gone: the Linux Standard Base specifies that its address is resolved via a call to the `__errno_location()` helper function [21].

5.7 Case study: auto async-signal safety

We have now described the role of *libgotcha*, and how *libinger* uses it to handle nonreentrancy. Before concluding our discussion, however, we note that *libgotcha* has other interesting uses in its own right.

As an example, we have used it to implement a small library, *libas-safe*, that transparently allows an application's signal handlers to call functions that are not async-signal safe, which is forbidden by POSIX because it is normally unsafe.

Written in 127 lines of C, *libas-safe* works by injecting code before `main()` to switch the program away from its starting libset. It shadows the system's `sigaction()`, providing an implementation that:

- Provides copy-based library isolation for signal handlers by switching the thread's next libset to the starting libset while a signal handler is running.
- Allows use of uncopyable code such as `malloc()` from a signal handler by deferring signal arrival whenever the thread is already executing in the starting libset, then delivering the deferred signal when the interruptible callback fires.

In addition to making signal handlers a lot easier to write, *libas-safe* can be used to automatically "fix" deadlocks and other misbehaviors in misbehaved signal-handling programs just by loading it via `LD_PRELOAD`.

We can imagine extending *libgotcha* to support other use cases, such as simultaneously using different versions or build configurations of the same library from a single application.

Operation	Duration (μs)
`launch()`	4.6 ± 0.05
`resume()`	4.4 ± 0.02
`cancel()`	4767.7 ± 1168.7
`fork()`	207.5 ± 79.3
`pthread_create()`	32.5 ± 8.0

Table 2: Latency of preemptible function interface

6 Evaluation

We now evaluate preemptible function performance, presenting several microbenchmarks and two examples of their application to improve existing systems' resilience to malicious or otherwise long-running requests. All experiments were run on an Intel Xeon E5-2683 v4 (Broadwell) server running Linux 4.12.6, rustc 1.36.0, gcc 9.2.1, and glibc 2.29.

6.1 Microbenchmarks

Table 2 shows the overhead of *libinger*'s core functions. Each test uses hundreds of preemptible functions, each with its own stack and continuation, but sharing an implementation; the goal is to measure invocation time, so the function body immediately calls `pause()`. For comparison, we also measured the cost of calling `fork()` then `exit()`, and of calling `pthread_create()` with an empty function, while the parent thread waits using `waitpid()` or `pthread_join()`, respectively.

The results show that, as long as preemptible functions are eventually allowed to run to completion, they are an order of magnitude faster than spawning a thread and two orders of magnitude faster than forking a process. Although cancellation takes milliseconds in the benchmark application, this operation need not lie on the critical path unless the application is cancelling tasks frequently enough to exhaust its supply of libsets.

Recall that linking an application against *libgotcha* imposes additional overhead on most dynamic symbol accesses; we report these overheads in Table 3a. Eager function calls account for almost all of a modern program's dynamic symbol accesses: lazy resolution only occurs the first time a module calls a particular function (Section 5.4) and globals are becoming rare (Section 5.6).

Table 3b shows that the *libgotcha* eager function call overhead of 14 ns is on par with the cost of a trivial C library function (`gettimeofday()`) and one-third that of a simple system call (`getpid()`). This overhead affects the entire program, regardless of the current libset at the time of the call. Additionally, calls to uncopyable functions from within a preemptible function incur several extra nanoseconds of latency to switch back to the main namespace as described in Section 5; Table 3c breaks this overhead down to show the cost of notification callbacks at the conclusion of such a call (always required by *libinger*).

6.2 Web server

To test whether our thread library could combat head-of-line blocking in a large system, we benchmarked *hyper*, the highest-performing Web server in TechEmpower's plaintext benchmark as of July 2019 [16]. The server uses *tokio-threadpool* for scheduling; because the changes described in Section 4 are transparent, making *hyper* preemptive was as easy as building against *libturquoise* instead. In fact, we did not even check out the *hyper* codebase. We configured *libturquoise* with a task timeout of 2 ms, give or take a 100-μs *libinger* preemption interval, and configured it to serve responses only after spinning in a busy loop for a number of iterations specified in each request. For our client, we modified version 4.1.0 of the *wrk* [14] closed-loop HTTP load generator to separately record the latency distributions of two different request classes.

Our testbed consisted of two machines connected by a direct 10-GbE link. We pinned *hyper* to the 16 physical cores on the NIC's NUMA node of our Broadwell server. Our client machine, a Intel Xeon E5-2697 v3 (Haswell) running Linux 4.10.0, ran a separate *wrk* process pinned to each of the 14 logical cores on the NIC's NUMA node. Each client core maintained two concurrent pipelined HTTP connections.

We used loop lengths of approximately 500 μs and 50 ms for short and long requests, respectively, viewing the latter requests as possible DoS attacks on the system. We varied the percentage of long requests from 0% to 2% and measured the round-trip median and tail latencies of short requests and the throughput of all requests. Figure 4 plots the results for three server configurations: `baseline` is cooperative scheduling via *tokio-threadpool*, `baseline+libgotcha` is the same but with *libgotcha* loaded to assess the impact of slower dynamic function calls, and `baseline+libturquoise` is preemptive scheduling via *libturquoise*. A 2% long request mix was enough to reduce the throughput of the *libgotcha* server enough to impact the median short request latency. The experiment shows that preemptible functions keep the tail latency of short requests scaling linearly at the cost of a modest 4.5% median latency overhead when not under attack.

6.3 Image decompression

The Web benchmark showed preemptive scheduling at scale, but did not exercise preemptible function cancellation. To demonstrate this feature, we consider decompression bombs, files that expand exponentially when decoded, consuming enormous computation time in addition to their large memory footprint. PNG files are vulnerable to such an attack, and although *libpng* now supports some mitigations [24], one cannot always expect (or trust) such functionality from third-party code.

We benchmarked the use of *libpng*'s "simple API" to decode an in-memory PNG file. We then compared against synchronous isolation using preemptible functions, as well as the naïve alternative mitigations proposed in Section 1. For preemptible functions, we wrapped all uses of *libpng* in a call to `launch()` and used a dedicated (but blocking) reaper thread to remove the cost of cancellation from the critical path; for threads, we used `pthread_create()` followed by `pthread_timedjoin_np()` and, conditionally, `pthread_cancel()` and `pthread_join()`; and for processes, we used `fork()` followed by `sigtimedwait()`, a conditional `kill()`, then a `waitpid()` to reap the child. We ran `pthread_cancel()` both with and without asynchronous cancelability enabled, but the former always deadlocked. The timeout was 10 ms in all cases.

Running on the benign RGB image `mirjam_meijer_mirjam_mei.png` from version `1:0.18+dfsg-15` of Debian's `openclipart-png` package showed `launch()` to be both faster and lower-variance than the other approaches, adding 355 μs or 5.2% over the baseline (Figure 5a). The results for `fork()` represent a best-case scenario for that technique, as we did not implement a shared memory mechanism for sharing the buffer, and the cost of the system call will increase with the number pages mapped by the process (which was small in this case).

Next, we tried a similarly-sized RGB decompression bomb from revision `b726584` of `https://bomb.codes` (Figure 5b). Without asynchronous cancelability, the pthreads approach was unable to interrupt the thread. Here, `launch()` exceeded the deadline by just 100 μs, a

Symbol resolution scheme	Time without *libgotcha* (*ns*)	Time with *libgotcha* (*ns*)
eager (load time)	2 ± 0	14 ± 0
lazy (runtime)	100 ± 1	125 ± 0
global variable	0 ± 0	3438 ± 13

(a) Generic symbols, without and with *libgotcha*

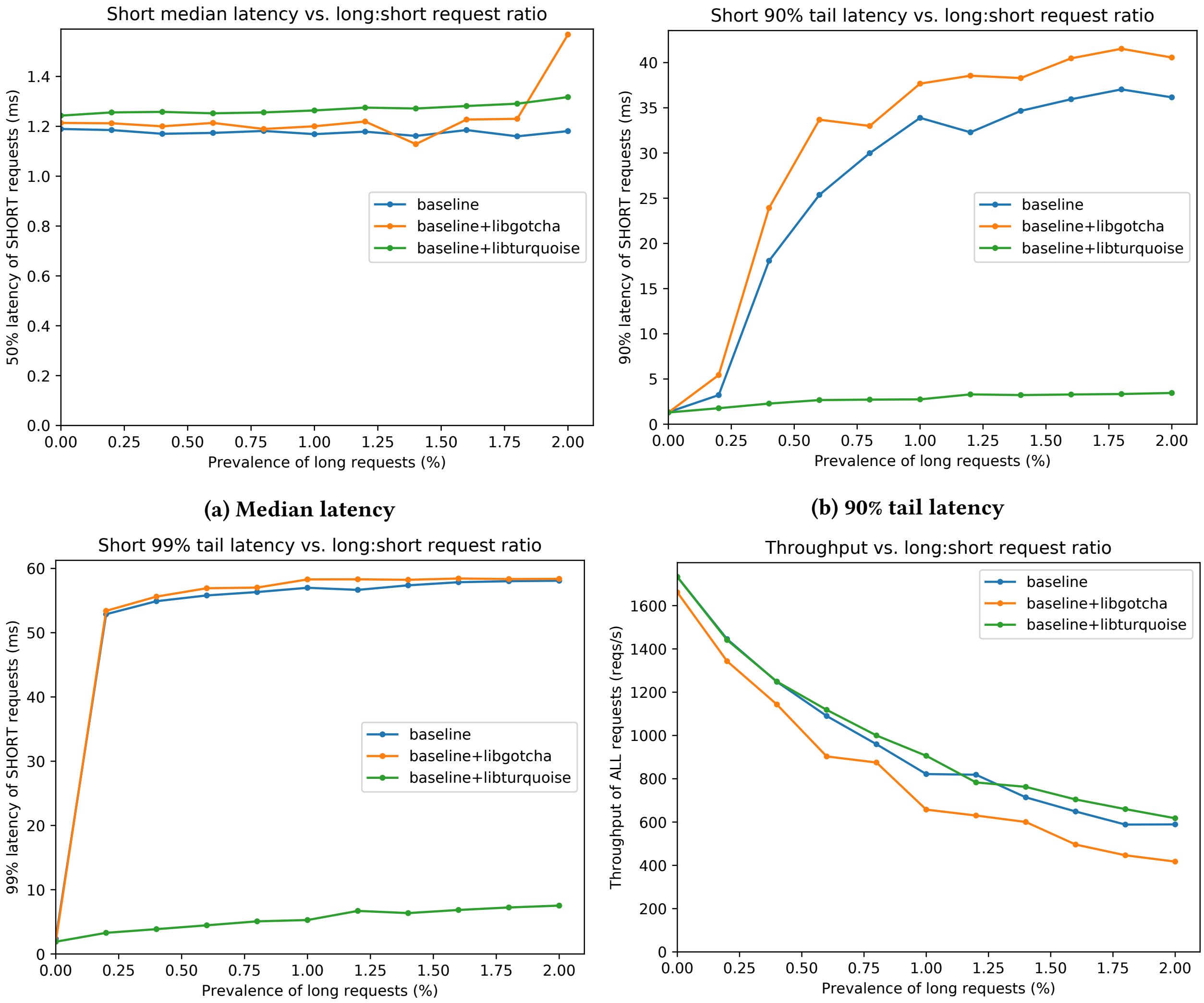
Baseline	Time without *libgotcha* (*ns*)
`gettimeofday()`	19 ± 0
`getpid()`	44 ± 0

(b) Library functions and syscalls without *libgotcha*

Trigger	Time with *libgotcha* (*ns*)
Uncopyable call	21 ± 0
Uncopyable call + callback	25 ± 0

(c) Uncopyable calls triggering a libset switch

Table 3: Runtime overheads of accessing dynamic symbols



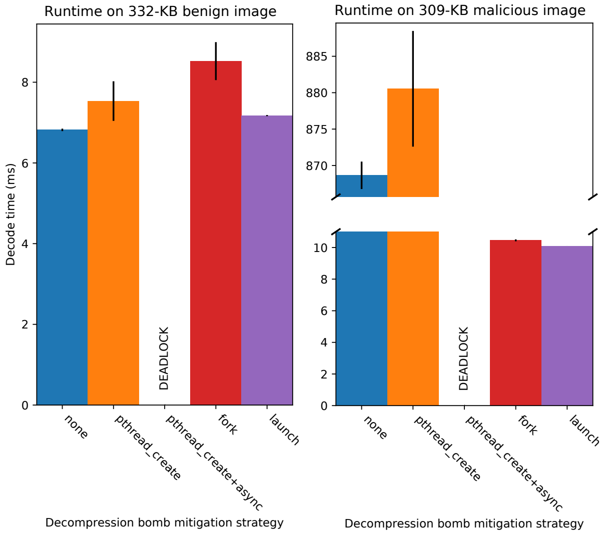
(a) Median latency

(b) 90% tail latency

(c) 99% tail latency

(d) Overall throughput

Figure 4: *hyper* Web server with 500-μs (short) and 50-ms (long) requests



(a) Benign image **(b) Malicious image**

Figure 5: *libpng* in-memory image decode times

figure that includes deviation due to the 100-μs preemption interval in addition to *libinger*'s own overhead. It again had the lowest variance.

Applying preemptible functions proved easy: the `launch()`/`cancel()` approach took just 20 lines of Rust, including the implementation of a reaper thread to move libset reinitialization off the critical path. In comparison, the `fork()`/`sigtimedwait()` approach required 25 lines of Rust. Note that both benchmarks include unsafe Rust (e.g., to use the *libpng* C library and zero-copy buffers).

7 Future work

One of our contributions is asynchronous cancellation, something rarely supported by the state of the art. In Section 3.5, we noted our lack of support for automated resource cleanup; however, we outlined a possible approach for languages such as Rust, which we intend to investigate further. Cancellation is currently our most expensive operation because of the libset reinitialization described in Section 5.3, but we plan to improve this by restoring only the writeable regions of each module.

Another area for improvement is signal-handling performance optimization: whereas *Shinjuku* is able to preempt every 5 μs with a 10% throughput penalty [17], we have observed a similar throughput drop while only preempting every 20 μs via our technique [6]. We have not yet heavily optimized *libinger*, and have reason to believe that doing so will allow our design to achieve a preemption granularity midway between those figures for the same throughput cost. Because *Shinjuku* executes in privilege ring 0, they preempt by issuing interprocessor interrupts (IPIs) directly rather than using Linux signals. Their microbenchmarks reveal an IPI:signal latency ratio of roughly 1:2.5 (1,993 vs. 4,950 CPU cycles), indicating that we are not achieving peak performance. Furthermore, a key design difference between our systems suggests that this ratio probably understates the performance we could achieve. In their benchmark, roughly 42% of cycles are spent sending each signal, a cost we can amortize because our design uses recurring timer signals to counter warmup effects. A further 6.9% of benchmarked cycles are spent propagating the signal between cores, which should not affect our system because we request the timer signals on the same core that will receive them rather than using a central watchdog thread to preempt all workers. Context switching is likely responsible for most of our unexpected latency: by writing our signal handler very carefully, we should be able to adopt the same optimizations they describe (skipping signal mask and floating-point register swaps).

The selective relinking technique that underlies our interface allows safe pausing and cancellation in the presence of shared state, *independent of preemption mechanism*. In lieu of a timeout, control transfer might result from another scheduling consideration, such as real-time computing or task priority.

8 Conclusion

We presented the lightweight preemptible function, a new composable abstraction for invoking a function with a timeout. This enabled us to build a first-in-class preemptive userland thread library by implementing preemption atop a cooperative scheduler, rather than the other way around. Our evaluation shows that lightweight preemptible functions have overheads of a few percent (lower than similar OS primitives), yet enable new functionality.

We believe the lightweight preemptible function abstraction naturally supports common features of large-scale systems. For example: In support of graceful degradation, a system might use a preemptible function to abort the rendering of a video frame in order to ensure SLA adherance. An RPC server might preserve work by processing each request in a preemptible function and memoizing the continuations; if a request timed out but was later retried by the client, the server could resume executing from where it left off.

Acknowledgements

This work was supported by the U.S. National Science Foundation under award CNS–1700521, as well as the Intel Science and Technology Center for Visual Cloud Systems. The authors thank Holden and the other members of the Pittsburgh Rust Coffee meetup for early implementation discussions about the safe use of unstructured control flow in Rust programs. We also thank our shepherd, Tim Harris, for his helpful suggestions.

References

[1] The Go programming language. https://golang.org, 2019.

[2] The Rust programming language. https://www.rust-lang.org, 2019.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM symposium on operating system principles (SOSP '91)*, 1991.

[4] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. In *HotOS '19: Proceedings of the workshop on hot topics in operating systems*, May 2019.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazères, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementaiton (OSDI'12)*, 2012.

[6] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference*, Boston, MA, 2018.

[7] A. Clements. Go runtime: tight loops should be preemptible. https://github.com/golang/go/issues/10958, 2015.

[8] T. Delisle. Concurrency in C∀, 2018. URL https://uwspace.uwaterloo.ca/handle/10012/12888.

[9] dlmopen. dlmopen(3) manual page from Linux man-pages project, 2019.

[10] U. Drepper. ELF handling for thread-local storage. Technical report, 2013. URL https://akkadia.org/drepper/tls.pdf.

[11] D. Eloff. Go proposal: a faster C-call mechanism for non-blocking C functions. https://github.com/golang/go/issues/16051, 2016.

[12] G. H. et al. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research Technical Reports, 2005. URL https://www.microsoft.com/en-us/research/publication/an-overview-of-the-singularity-project.

[13] GitHub. Tokio thread pool. https://github.com/tokio-rs/tokio/tree/tokio-threadpool-0.1.16/tokio-threadpool, 2019.

[14] GitHub. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk, 2019.

[15] C. T. Haynes and D. P. Friedman. Engines build process abstractions. Technical Report TR159, Indiana University Computer Science Technical Reports, 1984. URL https://cs.indiana.edu/ftp/techreports/TR159.pdf.

[16] hyper. hyper: Fast and safe HTTP for the Rust language. https://hyper.rs, 2019.

[17] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *Proc. 16th USENIX NSDI*, Boston, MA, Feb. 2019.

[18] libev. libev. http://libev.schmorp.de.

[19] libevent. libevent. https://libevent.org.

[20] libuv. libuv: Cross-platform asynchronous I/O. https://libuv.org.

[21] Linux Standard Base Core specification. __errno_location. http://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/baselib---errno-location.html, 2015.

[22] M. S. Mollison and J. H. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA, 2013.

[23] mordor. mordor: A high-performance I/O library based on fibers. https://github.com/mozy/mordor.

[24] PNG reference library: libpng. Defending libpng applications against decompression bombs. https://libpng.sourceforge.io/decompression_bombs.html, 2010.

[25] pthread_setcanceltype(3). pthread_setcanceltype() manual page from Linux man-pages project, 2017.

[26] sigaction. sigaction(2) manual page from the Linux man-pages project, 2019.

[27] signal-safety. signal-safety(7) manual page from Linux man-pages project, 2019.

[28] TerminateThread. TerminateThread function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread, 2018.

[29] The Rust Programming Language. Extensible Concurrency with the Sync and Send Traits. https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html, 2019.

[30] The Rust Reference. Behavior not considered unsafe. https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html, 2019.

[31] timer_create. timer_create(2) manual page from the Linux man-pages project, May 2020.

[32] C. J. Vanderwaart. Static enforcement of timing policies using code certification. Technical Report CMU-CS-06-143, Carnegie Mellon Computer Science Technical Report Collection, 2006. URL http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-143.html.

coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O

Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, Yaozu Dong
Intel Corporation

Abstract

Direct assignment of I/O devices (Direct I/O) is the best performant I/O virtualization method. However, it requires the hypervisor to statically pin the entire guest memory, thereby hindering the efficiency of memory management. This problem can be fixed by presenting a virtual IOMMU (vIOMMU). Emulation of its DMA remapping capability carries sufficient information about guest DMA buffers, allowing the hypervisor to do fine-grained pinning of guest memory. However, established vIOMMUs are not widely used by commodity guests due to the emulation cost, thus cannot reliably eliminate static pinning in direct I/O.

We propose and implement *coIOMMU*, a new vIOMMU architecture for efficient memory management with a cooperative DMA buffer tracking mechanism. The new mechanism provides a dedicated interface for hypervisor and guest to exchange DMA buffer information over a shared DMA tracking table (DTT), orthogonal to the costly DMA remapping interface. We also explore two techniques: smart pinning and lazy unpinning, to minimize the impact on the performance of direct I/O. Our evaluation results show that coIOMMU dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost. Moreover, the desired semantics of DMA remapping can be sustained when cooperative tracking is enabled alongside. Overall, we believe that coIOMMU can serve as a reliable solution for efficient memory management in direct I/O.

1. Introduction

Direct I/O [1, 21, 29, 31, 37, 39, 48, 49, 50] is the best performant I/O virtualization method and a cornerstone capability in data centers and clouds. It allows the guest to directly interact with I/O devices without the intervention from software intermediary. An I/O memory management unit (IOMMU) [3, 14, 16] helps prevent Direct Memory Access (DMA) attacks in direct I/O by providing the capability of *DMA remapping*. Each assigned device is associated with an IOMMU page table (IOPT), configured by the hypervisor in a way that only the memory of the guest that owns the device is mapped. The IOMMU walks the IOPTs to validate and translate DMA requests, achieving inter-guest protection among directly assigned devices.

Most devices do not tolerate DMA faults, implying that guest buffers must be pinned in host memory and mapped in the IOPT before they are accessed by DMAs. However, the hypervisor does not know which pages are mapped by the guest when it is eliminated from the direct I/O path. Consequently, it has to pin the entire guest memory upfront, a.k.a *static pinning* [7, 44]. This heavily hinders the efficiency of memory management and worsens memory utilization, as pinned pages cannot be reclaimed for other purposes.

Presenting a virtual IOMMU (vIOMMU) [8, 23, 29, 52, 60] to the guest allows *fine-grained pinning* of guest memory for efficient memory management, although its primary purpose is to help the guest protect itself against buggy drivers. The hypervisor emulates the DMA remapping interface by: 1) walking the virtual IOPT (vIOPT) to identify the affected buffers; 2) pinning and unpinning the buffers in the host memory; and 3) mapping and unmapping them in the physical IOMMU to enforce protection as desired by the guest. Naturally, the emulation leads to a fine-grained pinning scheme, if the guest always uses the vIOMMU to remap its DMA buffers.

Unfortunately, established vIOMMUs are not applicable as a reliable solution for *fine-grained pinning*. Their virtual DMA remapping capabilities are disabled by most guests [8, 24, 30, 38, 51] in typical usages such as public cloud, because significant emulation cost may be incurred due to frequent mapping operations in the guest. Such cost could be alleviated through side-core emulation [8] or para-virtualized extension [23, 52]. However, the side-core emulation requires an additional CPU core to perform the emulation; and can only achieve optimal performance with deferred IOTLB invalidation, leading to compromised security. Para-virtualized extension reduces the virtualization overhead with optimized interfaces, but it still involves large number of VM-exits at the time of guest DMA mappings/unmappings, hence limiting the performance. Therefore, they did not change the fact that established vIOMMUs are used only in limited circumstances, e.g. when intra-guest protection is valued over the overhead of DMA remapping.

We argue that mixing the requirements of protection and pinning, through the same costly DMA remapping interface, is needlessly constraining. Protection is a guest requirement, while pinning is for host memory management. The two do not always match, thus favoring one may easily break the other. Instead, we aim to provide a reliable solution for fine-grained pinning by decoupling it from protection.

We propose and implement a new vIOMMU architecture called coIOMMU, which helps the hypervisor achieve efficient memory management in direct I/O. It introduces a dedicated mechanism for cooperative DMA buffer tracking, orthogonal to the costly DMA remapping interface. coIOMMU allows the hypervisor and guest to communicate over a DMA tracking table (DTT) located in a shared memory region. The guest records the mapping status of its DMA buffers in the DTT and the hypervisor walks the DTT to identify the corresponding pinning requirement. coIOMMU further minimizes the number of notifications from the guest, with two optimizations: (1) *smart pinning*, which heuristically pins frequently used pages and timely shares its pinning status with the guest, to enable precise notification in guest-mapping operations; and (2) *lazy unpinning*, which asynchronously unpins guest pages to eliminate notifications in guest-unmapping operations. On the other hand, the new mechanism does not affect the desired semantics of DMA remapping. It can be enabled with or without DMA remapping, as a reliable and standard interface to achieve fine-grained pinning in direct I/O.

We implement coIOMMU by extending KVM/QEMU vIOMMU and Linux guest. The concept and implementation can be easily ported to other hypervisors, vIOMMUs and guest OSes. Overall, the main contributions of this paper are:

- Observing that established vIOMMUs cannot reliably fix the problem of static pinning in direct I/O, due to the costly DMA remapping interface.
- Proposing and implementing coIOMMU, the first vIOMMU that introduces a dedicated DMA buffer tracking mechanism for fine-grained pinning.
- Introducing smart pinning and lazy unpinning to dramatically reduce the tracking overhead in fine-grained pinning.
- Conducting comprehensive evaluations under different Linux protection policies, with benchmarks in direct networking, storage, and GPU.
- Demonstrating that coIOMMU not only dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost, but also sustains the desired security as required in specific protection policies.

The rest of the paper is organized as follows. The background and motivation are first provided in section 2. We present the design of coIOMMU in section 3 and its implementation in section 4. Finally, the evaluation results are shown and discussed in section 5, with future work and conclusion drawn in section 6.

2. Motivation

2.1. The Problem

Direct I/O is the best performant I/O virtualization method by enabling direct communication between the guest and the I/O devices. Removal of the software intermediary not only provides much better performance than other I/O virtualization approaches, but also allows faster time-to-market for virtualizing new I/O acceleration capabilities. Direct I/O proliferates via device-side virtualization. Single-Root I/O Virtualization (SR-IOV) [1, 13] allows the device to multiplex its resource into virtual functions, each independently assignable to a guest. Cloud service providers even offload para-virtualized backend drivers into directly assigned devices [11, 12]. With these hardware trends, direct I/O has gained mainstream support in commodity hypervisors and is becoming a cornerstone capability in data centers and clouds.

IOMMUs [3, 14, 16] are introduced by hardware vendors to prevent assigned devices from touching arbitrary memory locations. Use of the IOMMU leads to the static pinning problem due to two factors: (1) most I/O devices do not tolerate DMA faults, and (2) the hypervisor does not know how guest memory is used for DMA. The hypervisor has to pin the entire guest memory upfront, assuming that every guest page might be a DMA page. This heavily hinders the efficiency of memory management and worsens memory utilization, as pinned pages cannot be reclaimed for other purposes.

2.2. Existing Solutions

Previous studies generally tackle this problem in two directions: making the device support DMA page faults or exposing the DMA buffer information to the hypervisor through software approaches.

DMA page faults allow all kinds of memory optimizations that CPU page faults provide. The PCI-SIG standardizes the support of DMA page faults with Address Translation Service (ATS) and Page Request Service (PRS) [2]. It was originally introduced to simplify the programming model on GPUs [27, 41, 42] and now also starts to find its way into NICs [6] and FPGA [9]. However, the latency of handling DMA page faults is 3x-80x higher than that of handling CPU faults [6, 40]. Such long latency, up to hundreds of microseconds, demands a larger on-device buffer to hold in-flight requests and incurs higher device cost. Handling such long latency in all critical paths further complicates the device. Therefore, most commodity devices do not support DMA page faults, or partially support it only for selective workloads. With time, it may become a preferable way for fine-grained pinning, but not anytime soon.

Alternatively, researchers also look at software approaches to expose enlightened guest DMA information to the hypervisor. Knowing when a guest page is mapped or unmapped allows the hypervisor to pin or unpin it dynamically. Willmann et al. [44] evaluates several mapping strategies, revealing that a big performance penalty is incurred when blindly doing hypercalls to notify the hypervisor of every guest mapping/unmapping operation. Yassour et al. [7] dramatically reduces such notifications with a guest-side pin-down cache. However, it puts a complex eviction policy in the guest and provides no intra-guest protection.

Presenting a vIOMMU [23, 29, 60] also provides sufficient information for fine-grained pinning, as a result of emulating its DMA remapping capability for intra-guest protection. However, such emulation may incur significant cost, especially when frequent mapping operations are requested by the guest. To trade off performance and protection, modern OSes typically implement different policies about DMA remapping. For example, Linux [8, 24, 30, 38, 51] implements *strict*, *lazy* and *passthrough* policies. Although DMA remapping is used in *strict* and *lazy* policies, the *passthrough* policy simply disables it to gain best performance. Obviously, the guest cannot provide any DMA buffer information to the hypervisor when the *passthrough* policy is selected. Unfortunately, major guest Linux distributions choose *passthrough* as default and even allow different policies across devices.

Recent studies focus on reducing the cost of emulating DMA remapping in vIOMMU. Tang et al. [52] reduces the remapping overhead by reusing old mappings and delaying their removal, however, at the cost of compromised security. Side-core emulation [8] achieves 100% of 10Gbps line rate with a fully emulated vIOMMU, but with relaxed protection and increased total cost. The overhead of DMA remapping is also tackled on bare metal [24, 30, 38]. While these works generally apply to the guest OS as well, most of them have not been adopted by commodity OSes due to its intrusiveness. In a nutshell, the cost of DMA remapping is still notable in the guest today, leaving the capability disabled or even not exposed in most cloud and data center usages.

2.3. DMA Tracking vs. DMA Remapping

We prefer the vIOMMU approach for two reasons: 1) it supports both intra-guest protection and fine-grained pinning; and 2) DMA page faults are not widely supported by commodity devices. However, we want to go a different direction from previous studies – to enable fine-grained pinning without being encumbered by the intrinsic cost of DMA remapping.

We argue that mixing the requirements of protection and pinning, through the same costly DMA remapping interface, is needlessly constraining. Protection is a guest requirement and relies on the DMA remapping capability, while pinning is for host memory management and needs the capability of tracking guest DMA buffers. The two do not always match, thus favoring one may just break the other, if both are enabled through the same interface. For example, the hypervisor either must fall back to static pinning by assuming that most guests disable protection, or, adopt fine-grained pinning by forcing all guests to enable protection and bear with added cost.

What about inventing a separate DMA buffer tracking mechanism to the vIOMMU, without relying on any semantics of DMA remapping? Separating DMA tracking from DMA remapping allows us to tackle the pinning and protection problems in parallel. If the new tracking mechanism incurs negligible cost, we can expect most guests to always enable it and reliably provide necessary information for fine-grained pinning. If feasible, such an approach would make the vIOMMU as the portal of efficient memory management in future data centers and clouds.

3. Design

We propose coIOMMU, a new vIOMMU architecture that helps the hypervisor achieve efficient memory management in direct I/O. coIOMMU provides a dedicated DMA buffer tracking mechanism that adopts a shared memory interface for efficient communication between host and guest. The guest records the mapping status of its DMA buffers through a shared DMA tracking table (DTT), for the hypervisor to decide its pinning strategy. coIOMMU also introduces two optimizations: smart pinning and lazy unpinning, to dramatically reduce the performance impact when achieving fine-grained pinning.

3.1. Goals

We want the new DMA buffer tracking mechanism to meet these goals:

Orthogonal to DMA Remapping - Our solution should allow DMA buffer tracking and DMA remapping independently configured by the guest. The new tracking mechanism, once enabled, should consistently supply sufficient information for fine-grained pinning, regardless of how DMA remapping is configured to protect guest. Enabling of DMA buffer tracking should not affect the desired protection semantics of DMA remapping.

Low Cost - DMA buffer tracking should incur negligible cost. Otherwise, it faces the same challenge as in DMA remapping: if significant cost is observed, why would one enable it by default? We focus on the efficiency of DMA buffer tracking itself and have no intention to further optimize DMA remapping in this work. The original performance expectation under each guest protection policy is set as the baseline for comparing the cost of DMA buffer tracking in our evaluations.

Non-intrusiveness - We want our solution to minimize the changes in the guest software stack, as a primary factor to gain mainstream support in commodity OSes. Commodity OSes provide a generic DMA API layer [25, 43] to route

DMA mapping requests from device drivers to underlying DMA driver. DMA buffers can be tracked either in the DMA API layer or specific DMA driver. We did not choose DMA API because any change in such common framework usually takes a long time to be adopted by commodity OSes.

Wide Applicability - We prefer a solution that works with all kinds of I/O devices rather than requiring additional changes in hardware or device drivers. We also expect such a solution to make no assumption on any vendor specific characteristics, so it can be easily ported to different vIOMMUs, either emulated or para-virtualized.

Extensibility - The solution should be extensible to help address other limitations in memory management. For example, another challenge in direct I/O is about lively migrating the guest with assigned devices, which requires the ability of tracking the pages that are dirtied by DMAs [20, 26, 28, 35]. We expect our solution can play as a portal of tracking all kinds of DMA buffer status for efficient memory management.

3.2. Architecture

The coIOMMU architecture is illustrated in Figure 1, composed of coIOMMU backend in hypervisor and coIOMMU driver inside the guest. The coIOMMU backend includes three main components: (1) DMA remapping engine (*remapEngine*), the same functionality for intra-guest protection as in established vIOMMUs, over a set of per-device vIOMMU page tables (vIOPTs); (2) DMA tracking engine (*trackEngine*), a new function dedicated for tracking guest DMA buffers over a global DMA tracking table (DTT); and (3) Page-pinning manager (*pManager*), which uses the information gathered by trackEngine to intelligently manage the pinning requirements of guest memory. The remapEngine and trackEngine are independently enumerated and managed by the coIOMMU driver, while pManager is hidden and activated automatically when trackEngine is enabled.

In our prototype, we build coIOMMU by extending an existing vIOMMU, which emulates the Intel VT-d hardware [3]. This allows us to focus on the new trackEngine and pManager, while inheriting the established DMA remapping logic as remapEngine. However, we make no assumption on the specific hardware or vIOMMU type. The design of trackEngine and pManager can be easily ported to any emulated or para-virtualized vIOMMU.

The trackEngine holds the base address of the DTT, which is allocated and registered by the coIOMMU driver. The format of the DTT is a hierarchical page table, containing the mapping information required by fine-grained pinning. trackEngine also includes a doorbell register to notify the hypervisor if necessary. Within the coIOMMU backend, trackEngine provides interfaces for pManager to access the DTT and also notifies pManager when the doorbell is rung. With this design, trackEngine acts as a standard interface solely for conveying the DMA information, while pManager actually uses the information to achieve fine-grained pinning. The separation between these two components allows coIOMMU to be easily extended for other purposes, e.g. by introducing another agent to track dirty pages, alongside pManager, while reusing the same trackEngine interface.

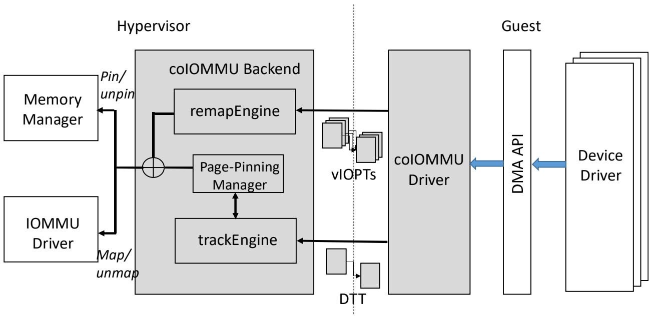


Figure 1: The architecture of coIOMMU

The coIOMMU driver intercepts the DMA API operations in the guest and updates the DTT accordingly. Modern OSes all implement a generic DMA API layer [25, 43], connecting device drivers to the underlying DMA driver to prepare their DMA buffers. The coIOMMU driver registers itself as a DMA driver to capture the latest mapping status of guest DMA buffers. This driver also enforces the desired protection semantics, as other vIOMMU drivers normally do today. In this way, DMA tracking is enabled without any change to the DMA API layer or specific device drivers of the guest.

The pManager contains hypervisor-specific policies for fine-grained pinning. A specific implementation may even include multiple policies and let the hypervisor dynamically choose a policy at runtime. We demonstrate two optimizations in §3.4: smart pinning and lazy unpinning, to minimize the notification overhead. When required, pManager talks to the memory manager for pinning or unpinning a set of guest pages and request the IOMMU driver for mapping or unmapping them in the physical IOPT. When both remapEngine and pManager are enabled, their pinning decisions are ORed together to favor the stricter requirement. Once a guest page is unpinned and unmapped, it can be reclaimed under whatever policy applied by the memory manager.

3.3. DMA Tracking Table (DTT)

The DTT records the mapping status of guest DMA buffers. It is shared by all assigned devices because the hypervisor only wants to know the DMA buffers of the entire guest. It is not necessary to track DMA buffers for virtual devices, assuming their DMAs are emulated by and already known to the hypervisor. The DTT is allocated by the guest, starting as empty and then filled dynamically according to intercepted DMA operations. We choose to track two categories of guest pages in the DTT: 1) the pages that are currently mapped by the guest and 2) the pages that have been unmapped but still pinned by the hypervisor. The latter category is necessary for lazy unpinning introduced in the next section.

One may argue why inventing a new table instead of reusing the vIOPTs, when the latter also carry the information of guest DMA buffers. We considered this approach but gave up for several reasons. First, the vIOPT is designed for intra-guest protection which disallows pinning a page after it is unmapped thus also negates lazy unpinning. Second, the table is indexed by guest I/O Virtual Address (IOVA) for the remapping purpose. The hypervisor has to walk every vIOPT to find out whether a guest page is mapped, which is too costly. Last but not the least, the format of vIOPT is typically vendor-specific, so extending it may not lead to good portability.

The DTT is a 4-level page table in 4KB pages, as shown in Figure 2. The 4KB leaf page consists of 512 DTT PTEs (DTEs) and each 8-bytes DTE is further split into 8 tracking units (TU). Each TU corresponds to one 4KB guest page. In total, the DTT can support up to 51-bits (9+9+9+9+3+12=51) guest physical address width, big enough for prevalent virtualization usages. Such design leaves 8-bits available in each TU. coIOMMU currently uses 3 bits for fine-grained pinning, with the other 5-bits reserved for future extension:

- 'M (mapped)', indicating a page currently mapped by guest for DMA. It is set and cleared by the guest before and after the corresponding DMA and is read-only to the hypervisor. This bit conveys the primary information used by fine-grained pinning.
- 'P (pinned)', marking a page currently pinned by the hypervisor. It is updated by the hypervisor to reflect the pinning status and is read-only to the guest, necessary for smart pinning.
- 'A (accessed)', telling whether a page has ever been used for DMA. The guest sets this bit alongside the setting of M-bit ('mapped' bit). Then it stays sticky until the hypervisor clears it in lazy unpinning.

An entry with both M and P bits cleared marks the page as invalid. If every entry of a DTT page is invalid, the guest may choose to free this page to save space.

3.4. Fine-grained Pinning

Two techniques are introduced in coIOMMU: *smart pinning* and *lazy unpinning*, to minimize the notification overhead of fine-grained pinning. We focus on the scenario where the DMA remapping capability of coIOMMU is disabled by the guest. In this case, there is no intra-guest protection requirement thus the hypervisor can pin more pages than what guest actually maps.

3.4.1. Smart Pinning

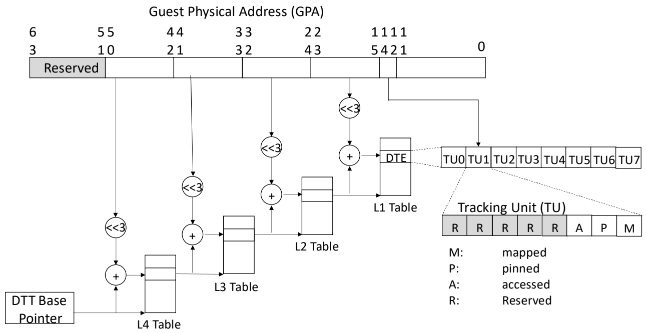


Figure 2: the format of the DTT

coIOMMU manages the pinning of guest pages in three ways: (1) *instantly pinning*: the guest instantly notifies the hypervisor to pin pages when they are being mapped, for correctness; (2) *precise notification*: the guest notifies the hypervisor if and only if the to-be-mapped pages are not pinned, to minimize the notification overhead; and (3) *speculatively pinning*: pManager heuristically pins the frequently used pages for performance.

First, pinning must be *instantly* done before any mapped page is used for DMA, because most devices do not tolerate DMA faults, as aforementioned. In such circumstance, the hypervisor must be notified by the guest to complete the pinning action in a timely manner, if the page has not yet been pinned.

Second, coIOMMU exposes the pinning status to the guest through the P-bit ('pinned' bit) in the DTT, for *precise notification*. If the P-bit is cleared by the hypervisor, the guest must notify the hypervisor instantly when mapping a page. Otherwise, no notification is needed at all. This optimization allows the guest to skip most notifications in its mapping operations.

Last, pManager *speculatively* selects and pins frequently used pages by leveraging the guest DMA locality, which has been identified in both previous studies [7, 44, 51] and our evaluation. The DTT includes an A-bit ('accessed' bit) to mark a page ever used for DMA. The guest sets the A-bit when mapping a page and leaves it set until the hypervisor clears it. pManager determines the ages of unmapped pages by periodically scanning the A-bits (and clears it after a scan). Young pages (with A-bit set) are candidates of frequently used pages and might be accessed soon again. So pManager heuristically pins them to avoid the overhead of another pinning notification in the near future.

Our evaluation shows that precise notification and speculative pinning can dramatically reduce the notification overhead in instant pinning by up to 99.9992% (from 1.5M to 11 notifications, per second), when running memcached with a 40Gbps NIC connection. One notification takes ~2000-4000 cycles in our evaluation, so 1.5M notifications per second may eat up 1-2 CPU cores without such optimization.

3.4.2. Lazy Unpinning

The pManager *lazily* unpins guest pages to completely eliminate the notification overhead in guest unmapping operations. It asynchronously scans the DTT to find out the pages that are unmapped but still pinned, and then unpins them in a batch. In our prototype, we process *lazy unpinning* and *speculative pinning* together in the same thread. Unpinned pages are reclaimable by the memory manager to increase overall memory utilization. In the same example of memcached, lazy unpinning eliminates another 1.5M notifications per second for guest unmapping operations, which means saving another 1-2 CPU cores, with the cost of pinning additional ~1% memory (0.32MB) than the total size of mapped pages (34.68MB), in average.

3.5. Intra-Guest Protection

The DMA remapping engine (remapEngine) can achieve fine-grained pinning as well, as it is required to precisely map and pin DMA buffers per guest protection requirements. However, one cannot solely rely on DMA remapping because the guest may selectively turn it off for certain devices according to its protection strategy. We describe two examples as below.

First, the guest may dynamically enable/disable DMA remapping for an assigned device, leaving the hypervisor to switch back and forth between static pinning and fine-grained pinning. For example, guest Linux typically enables DMA remapping when assigning a device to its user space and then disables remapping when returning the device back to its kernel space [45]. The switch between static and fine-grained pinning may lead to intermittent out-of-memory errors in a budget system. Moreover, the hypervisor needs to unpin all the guest pages when switching away from and then re-pin them when switching back to static pinning, leading to increased overhead.

Second, if the guest enables DMA remapping only for selected devices, DMA remapping cannot provide full DMA buffer information for fine-grained pinning. For example, most Linux distributions enable DMA remapping only for untrusted devices, based on physical characteristics of the device [61, 62]. Such flexible configuration is possible because DMA remapping is typically enabled per device. However, fine-grained pinning needs to know DMA buffers used by all assigned devices in the guest, even for the ones that are not protected with DMA remapping. In such case, the hypervisor must fall back to static pinning with reduced memory utilization.

In both of these examples, DMA buffer tracking of coIOMMU allows reliably providing full DMA buffer information to enable fine-grained pinning. When tracking and remapping are both enabled, it is possible for the two to make different pinning decision for the same page. In such case, the decision from the DMA remapping interface takes precedence, because we must not break any protection semantics desired by the guest.

4. Implementation

We implement coIOMMU by extending the virtual Intel VT-d, which is an emulated vIOMMU in QEMU [58] (the device model of KVM hypervisor [10]), and the intel-iommu driver in the guest Linux. In QEMU, the original DMA remapping logic of the virtual VT-d is reused as remapEngine, while trackEngine and pManager are developed from scratch. Guest-side changes are all contained in the intel-iommu driver and hidden behind the Linux DMA API layer. There is no change required in guest device drivers. Currently, coIOMMU adds ~700 LOC in QEMU and ~1000 LOC in guest.

coIOMMU driver - coIOMMU driver extends guest intel-iommu driver to manage the trackEngine when the capability is detected. The intel-iommu driver registers callbacks to the Linux DMA API layer for mapping and unmapping DMA pages in different forms, e.g. for single page or scatter-gathered page list, for pre-allocated pages or newly allocated pages, etc. We extend the driver by extracting the DMA buffer information from those callbacks and updating the corresponding tracking units (TUs) in DTT. The DTT is allocated in the guest memory, which is always accessible by the commodity KVM hypervisor. If such direct access is prohibited in some specific security related usage cases [55, 56], the DTT should be allocated in a shared memory region. Last, the coIOMMU driver conditionally notifies the hypervisor based on the DTT status.

trackEngine - We extend the virtual VT-d with several changes: (1) a capability bit for enumerating the presence of trackEngine, (2) an enabling bit for activating trackEngine, (3) a register holding the base address of the DTT, (4) a register as the doorbell interface for triggering notification to pManager, and (5) a register pointing to the base address of the notification structure. The notification structure is designed to allow batching requests of multiple pages into one notification, in case of those pages are mapped together. trackEngine also provides function calls for pManager to scan and update the DTT.

pManager – The implementation of pManager can be split into two parts. First, it provides direct function calls for trackEngine to complete instant pinning. The functions are invoked synchronously in the vCPU threads when QEMU emulates the guest write to the doorbell register. Second, pManager also launches a thread for lazy unpinning and speculative pinning, woken up every one second. This thread scans the DTT to find out all the pages that are unmapped but still pinned and speculatively unpin them based on their A-bits. When a pinning decision is made, pManager invokes the VFIO API

[45] to pin/unpin selected pages and map/unmap them in the IOMMU.

Sub-Page Mappings - Multiple DMA buffers may co-locate in the same 4KB guest page, e.g. as widely observed when handling network packets. Sub-page mappings imply that one page might be mapped and unmapped multiple times. In such case, coIOMMU driver tracks the mapping count of each mapped page and clears the "M-bit" of the corresponding entry only when its count reaches zero. We choose to leverage the 5 reserved bits in each TU as the mapping count, holding up to 31 sub-page mappings. Doing so simplifies the implementation and works well in our evaluations. Other implementations may choose different structures for such tracking purpose.

Concurrency - coIOMMU must properly handle concurrent pinning/unpinning requests between multiple vCPU threads and the unpinning thread, as shown in Figure 3.

First, multiple vCPUs may try to map and pin the same DMA page simultaneously, e.g. in sub-page mapping scenario. We employ different locking mechanisms in guest and host for race avoidance. Within the guest kernel, spinlock is required for atomically setting the 'mapped' flag and checking the 'pinned' status of a target page. It is necessary as DMA mappings may happen in the guest interrupt context. On the other hand, a mutex is introduced in QEMU for atomically completing the actual pinning actions: 1) rechecking the 'pinned' status; 2) pinning the page; and 3) updating the 'pinned' flag.

Second, race condition may happen between concurrent pinning requests (from the vCPU threads) and unpinning requests (from the unpinning thread). For example, it is possible seeing an unpinning operation starts before, yet completes after, an in-flight pinning request. Such race may lead to the pinning request completing successfully but with the target page actually unpinned. We introduce two mechanisms to solve this problem. For one, the unpinning thread needs to check the 'mapped' flag before and after clearing the 'pinned' status. We call this special sequence as double-detection, necessary to catch in-flight change of the mapping status in the guest side. For two, the unpinning thread also needs to acquire the aforementioned QEMU mutex for completing its unpinning actions. In particular, the second check of the 'mapped' flag must be done with the mutex acquired and before conducting the unpinning action. If the 'mapped' status becomes true, indicating that a pinning action is in progress for the target page, the unpinning thread should cancel the unpinning operation immediately.

4.1. Discussion

Applicability - coIOMMU applies to all kinds of directly assigned devices, without the need of ad-hoc changes in hardware or software. Porting our Linux implementation to a new guest OS is straightforward, as long as the OS implements a generic DMA API layer which, obviously, is already a common feature in commodity OSes today. On the other hand, the implementation of trackEngine and pManager is vendor-neutral and self-contained. The separation between DMA tracking and DMA remapping allow coIOMMU implementation to be easily portable to other vIOMMUs, regardless of whether remapEngine is emulated or para-virtualized.

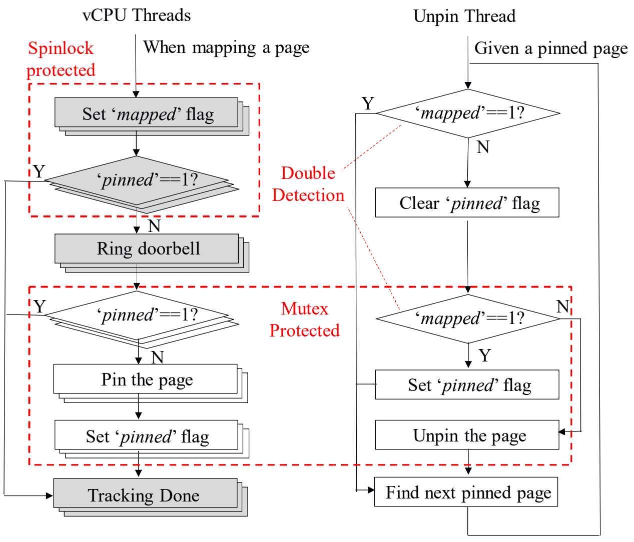


Figure 3: Race avoidance between concurrent pinning and unpinning operations. Gray boxes are guest actions, and white are host.

Extensibility - The page table format of the DTT can be extended to address other limitations in memory management. For example, introducing a "D (dirty)" bit in the TU provides a generic solution for tracking dirty pages when lively migrating VMs in direct I/O. Similarly, using a "W (writable)" bit to indicate read-only page enables the hypervisor to implement copy-on-write features. Ideally, a specific implementation may extend the DTT to include the same set of permission or status bits as available in a CPU page table.

Currently the DTT tracks DMA buffers in 4KB granularity. It is sufficient for most direct I/O usages, as DMA buffers are typically allocated in scattered 4KB pages. When large DMA buffer is used, we rely on pManager to merge batched pinning requests on continuous DMA pages into 2MB-based requests. We observed such optimization leads to ~4.5% FPS improvement in direct GPU benchmark, as illustrated in 5.1. Alternatively, one may also directly extend the DTT format to support 2MB-granular tracking entries.

Kernel Bypassing - coIOMMU also applies to various kernel bypassing techniques [32, 33, 45], which allow applications to directly manage DMA buffers in user space. Applications are untrusted, so they must first register a trunk of memory to the kernel and then manage within that trunk. The registration goes through proper kernel interfaces, e.g. AF_XDP [33] or VFIO [45] in Linux, which finally call into the coIOMMU driver for actual mappings and unmappings thereby are still tracked in the DTT. Kernel bypassing may increase the memory footprint because applications usually register a one-off big buffer pool to avoid calling into the kernel frequently. We leave optimizing such workloads as future work.

DMA Page Faults – For devices which do support DMA page faults, on-demand memory allocation/reclaim can happen at any time thus one could implement fine-grained pinning without using coIOMMU. However, coIOMMU may still provide two benefits in such circumstance. First, the overhead of handling DMA page faults might be non-negligible in hot data paths. coIOMMU allows the guest to reduce the number of faults by proactively requesting pre-pinning of hot pages, based on the knowledge that is easily extracted from DTT, yet invisible or difficult to acquire in legacy host. Second, some devices may allow DMA page faults only in selective data paths. Hypervisor could enable coIOMMU alongside the fault-based pinning scheme, to track DMA pages which are touched in non-faultable data paths in such devices.

Guest Cooperation - coIOMMU is a para-virtualized approach thus requires guest cooperation. We plan to submit our work to Linux and QEMU community, so coIOMMU could be enabled by default in most Linux distributions in the future. However, it is possible that a selfish guest may deliberately report fake DMA pages or simply disable coIOMMU driver to get more pages pinned than a cooperative guest. When required, one may choose to build a quota mechanism alongside the new tracking interface of coIOMMU. For example, the memory ballooning mechanism [57] can be extended to convey the quota information of both total memory and DMA memory, based on the service level agreement of the guest. Afterward, pManager could reject new pinning requests from any guest after its quota is exceeded.

5. Evaluation

Our evaluation aims to answer several questions. How does the overhead imposed by coIOMMU compare to that of established vIOMMUs? How many pages are pinned in various direct I/O usages when using coIOMMU to enable fine-grained pinning? Does coIOMMU sustain the desired performance and security under different intra-guest protection policies? We answer these questions by planning our evaluation to focus on four aspects: footprint, overhead, security and applicability.

Evaluated Modes - We evaluate six modes as shown in Table 1. The guest intel-iommu driver supports three protection policies: 1) *passthrough*, the default policy that disables DMA remapping for performance; 2) *strict*, using DMA remapping to gain full protection; and 3) *lazy*, trading off some security for performance when using DMA remapping (e.g. by deferring and batching IOTLB invalidations). We study the three policies for coIOMMU and a state-of-the-art vIOMMU, respectively, thus leading to six modes in total. In our prototype, coIOMMU inherits the DMA remapping logic of the virtual VT-d, so we choose this emulated vIOMMU solution to represent state-of-the-art vIOMMUs for fair comparison. We use {*PT-O*, *ST-O, LA-O*} to indicate the three protection policies with virtual VT-d and {*PT-N*, *ST-N*, *LA-N*} for the policies with coIOMMU. 'O' stands for the 'old' emulated VT-d while 'N' represents the 'new' coIOMMU.

mode	abbr.	DMA remapping	DMA buffer tracking	pinning model	protection
passthrough (virtual VT-d)	*PT-O*	unused	n/a	static	no
passthrough (coIOMMU)	*PT-N*	unused	used	fine-grained	no
strict (virtual VT-d)	*ST-O*	used	n/a	fine-grained	full
strict (coIOMMU)	*ST-N*	used	used	fine-grained	full
lazy (virtual VT-d)	*LA-O*	used	n/a	fine-grained	relaxed
lazy (coIOMMU)	*LA-N*	used	used	fine-grained	relaxed

Table 1: Evaluated modes in coIOMMU and virtual VT-d

Experimental Setup - Our setup consists of three machines, all running Ubuntu 16.04 with kernel 5.0.0. The primary machine, used for networking and storage tests, is equipped with a 16-core Intel Xeon Cascade Lake CPU at 2.7GHz, one 64GB DDR4 DIMM, an Intel XL710 40Gbps NIC, and two Intel 760P series 1TB NVMe SSDs. The 2nd machine acts as the network traffic generator, with another XL710 NIC connected to the primary machine back-to-back. It includes dual Intel Xeon Gold 6140 CPUs, each with 18 cores at 2.30GHz and 64GB DDR4 memory. The last machine is used for GPU evaluation, equipped with Intel Core i7-7567U CPU with four cores at 3.50GHz, 32GB DDR4 memory, a 256GB Intel 520 series SSD, and an Intel® Iris® Plus graphics 650 GPU.

The VM of the first machine is based on RHEL7.2 with kernel 5.1.0-rc3+, configured with 16 vCPUs, 32GB memory, and a directly assigned device – either a XL710 NIC or a 760P SSD, according to whether direct-networking or direct-storage is under evaluation. The two assigned devices are enabled independently, to avoid mutual interference from section 5.1 to section 5.5. In section 5.6, we evaluated their performance running combined workloads with both devices assigned. The VM for direct GPU includes Ubuntu 18.04 with kernel 5.1.0-rc3+, 4 vCPUs, 4GB memory, and a directly assigned Intel® Iris® Plus graphics 650 GPU. The vCPUs of both VMs are 1:1 pinned to the physical cores for stable results.

Benchmarks - We choose both micro-benchmarks and macro-benchmarks for evaluating the six modes in direct networking, direct storage and direct GPU:

- *Netperf* [63] is a standard micro-benchmark to measure networking throughput. We perform Netperf stream receive (RX) and transmit (TX) tests, using 64KB message size with 16 Netperf client/server instances (one per core) in the guest. Aggregated throughput is reported.

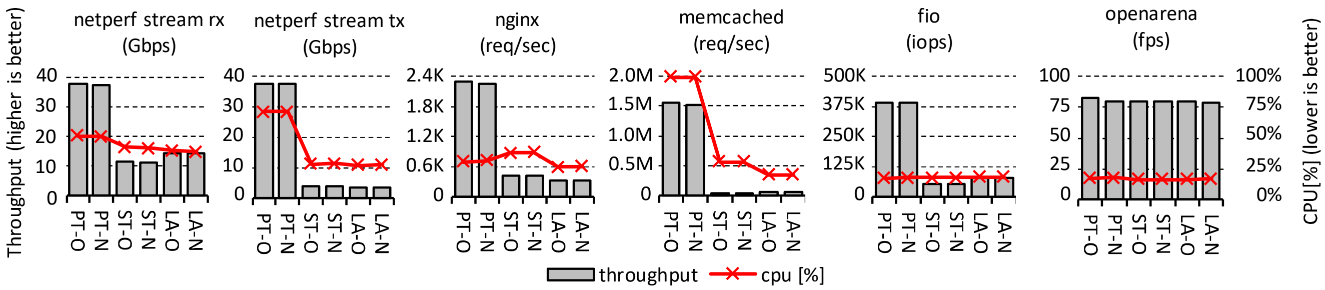


Figure 4: Performance of the six modes (100% CPU is 4 cores in openarena, and 16 cores in all other benchmarks)

mode	netperf stream rx		netperf stream tx		nginx		memcached		fio		openarena	
	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits
PT-O	352,224	0	577,037	0	525,974	0	3,110,716	0	781,055	0	44	0
PT-N	348,335	2,379	572,136	415	525,849	115	3,039,414	11	780,186	9	44	22
ST-O	109,403	109,403	64,448	64,448	72,239	72,239	104,354	104,354	109,864	109,198	44	44
ST-N	108,607	108,607	64,352	64,352	71,682	71,682	103,984	103,984	107,948	107,948	44	44
LA-O	141,844	71,013	59,645	29,896	63,230	31,702	145,309	72,744	163,085	81,655	44	23
LA-N	141,572	70,883	58,398	29,273	62,569	31,370	144,690	72,434	162,417	81,322	44	23

Table 2: The average number of completed DMA operations vs. incurred VM exits, per second.

- *Nginx* [64] is a high-performance HTTP web server. We use ApacheBench [69] to measure the number of concurrent requests that Nginx server can serve. We run ApacheBench to issue 16 concurrent requests of a static 1MB file, through the Nginx server installed in the guest.
- *Memcached* [65] is a popular in-memory key-value store, usually benchmarked using memaslap [70]. We use the default memaslap configuration with 64-byte keys, 1KB values, and 90%/10% GET/SET operations. In the VM, we launch 16 memcached instances driven by 16 memaslap threads each issuing 8 concurrent requests.
- *fio* [66] is a standard micro-benchmark to measure disk performance for wide range of storage types. We configure 16 fio threads, each performing asynchronous direct random reads from the assigned SSD, in 512-byte blocks and 128 in-flight requests.
- *OpenArena* [67] is a 3D first-person shooter game, used to benchmark direct GPU. The throughput is reported in frame-per-second (fps).

In addition, we also selectively run sysbench [68] as a memory benchmark and DPDK [32] for user-space networking stack, for specific evaluation purposes.

5.1. Overhead

We record the performance of aforementioned benchmarks in each evaluation mode, as shown in Figure 4. CPU utilization is aggregated over all cores, i.e. one core at 100% CPU would be reported as 100%/4=25% CPU utilization with 4 cores (for OpenArena) or 100%/16=6.25% CPU utilization with 16 cores (for all other benchmarks). In addition, we also capture the per-second number of completed DMA operations and associated VM-exits when running those benchmarks, in Table 2. All benchmarks run 30 seconds, except OpenArena, which must run to end in around 42 seconds. Next, we compare coIOMMU to virtual VT-d under the three Linux protection policies, respectively.

Passthrough - All networking benchmarks (left four in Figure 4) exhibit consistent results under the passthrough policy: coIOMMU (*PT-N*) retains the performance comparable to that of the virtual VT-d (*PT-O*), with less than 3% throughput degradation and negligible variation in CPU utilization. Such low cost is further explained in Table 2 – although hundreds of thousands of DMA operations are tracked per second, the majority of them do not trigger any VM-exit to notify the hypervisor, due to the optimization of *smart pinning* and *lazy unpinning*. For example, the lowest VM-exit number is observed in memcached, with only 11 VM-exits incurred by ~3M DMA operations.

The overhead of coIOMMU is unrecognizable in FIO but incurs 4.5% FPS drop in OpenArena. We found that OpenArena maps a big buffer (~240MB) in a batch at its launch time, with many pages adjacent to each other. In such case, pinning the buffer in 2MB size is more efficient than pinning in 4KB size, due to increased IOTLB efficiency. Unfortunately, 2MB pinning is not supported in our initial coIOMMU implementation, while it is the preferred option when KVM statically pins the entire guest memory in *PT-O*. After coIOMMU was extended to also conduct 2MB pinning for OpenArena, it then reaches the same performance as the virtual VT-d (not shown in the figure). We do not enable huge page pinning in other benchmarks, because they are observed with frequent mapping operations on many scattered 4KB pages. Blindly doing

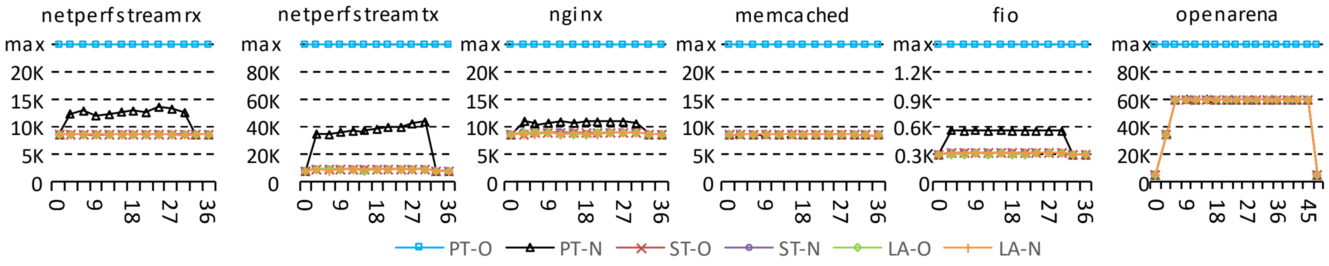


Figure 5: The number of pinned pages sampled in 3 second interval, taken from the beginning of the benchmarks to 6 seconds after their completion. 'max' indicates the total pages of guest memory.

huge page pinning simply adds more cost and footprint in those circumstances.

Strict and Lazy - We did not observe recognizable difference between coIOMMU (*ST-N* and *LA-N*) and virtual VT-d (*ST-O* and *LA-O*) in all benchmarks, regarding to both throughput and CPU utilization. There are much fewer DMA operations completed in the *strict* and *lazy* policy than that in the *passthrough* policy, due to the emulation cost of DMA remapping. As shown in Table 2, the reduction is between 2.46x (in Netperf RX) to 29.8x (in memcached) in all evaluated benchmarks. The tracking overhead in coIOMMU is negligible when comparing to the overhead of DMA remapping.

We also explore an interesting finding between *lazy* and *strict* in Figure 4, although not directly related to coIOMMU. It is a common learning that batching IOTLB invalidations generally brings better performance than strictly invalidating the IOTLB one-by-one. However, it is not always the case in virtualization – we observed 11% and 23% lower throughput when comparing *lazy* to *strict* in Netperf TX and Nginx. We find the batching interface of the virtual VT-d is the root cause. Its emulation requires walking the entire vIOPT to identify every valid mapping. If the walking cost exceeds the cycles of saved invalidations, the performance of *lazy* is instead worse than that of *strict*. We leave studying more efficient batching interface and policy for another research.

5.2. Memory Footprint

We sample the number of pinned pages every 3 seconds, from the beginning of the benchmarks to 6 seconds after its completion, in Figure 5. The extra 6 seconds are used to evaluate the elasticity of the six modes, against transitional system business. One note – the '*max*' mark in the Y-axis indicates the total number of guest pages, representing the case of static pinning. It is 8M (for 32GB memory) in most benchmarks and 1M (for 4GB memory) in OpenArena.

All six modes exhibit the same pattern in all benchmarks, except *PT-N*. First, *PT-O* is tied to static pinning, thereby always sitting in the top 'max' location. Second, all four modes with DMA remapping enabled (*ST-O*, *ST-N*, *LA-O*, and *LA-N*) pin the least number of pages, because they need strictly follow the desired protection semantics. As such, their lines completely overlap in each diagram in Figure 5. The line of *PT-N* (coIOMMU in the passthrough policy) fluctuates in the middle due to smart pinning, which heuristically pins guest pages for balancing performance and footprint. So, it is the focus of our following analysis.

Networking - All four networking benchmarks (left four in Figure 5) start and end with the same number of pinned pages (~8800 pages) in *PT-N*. Those always-pinned pages come from Intel i40e NIC driver, which pre-maps 512 pages per vCPU as the receive buffer pool when the NIC is enabled. The number sums up to 8192 pages with 16 vCPUs in our configuration.

The largest footprint is observed in Netperf stream TX, with up to 44530 pinned pages (174MB). It is ~4.4x of the pages that are actually mapped for DMA at that time. The additionally pinned 34158 pages reflect the DMA temporal locality, occupying only 0.4% of the total 32GB guest memory. coIOMMU recognizes such locality thus sustains the performance of static pinning when keeping a small memory footprint. Netperf stream RX pins fewer pages (up to ~18000) than TX, due to better DMA temporal locality – Intel i40e NIC driver prefers to use the pre-mapped 8192 pages for incoming packets. On the other hand, Nginx and Memcached are less throughput sensitive than Netperf TX/RX, yielding a transfer rate of 2.3Gbps and 1.34 Gbps respectively. Accordingly, there are fewer pages used for DMA in the two benchmarks, leading to smaller footprint in coIOMMU.

Storage - We configure fio to perform asynchronous direct random reads from the assigned SSD, to avoid page cache and readahead optimization in guest Linux. 16 fio threads are launched to read the disk with a 512-byte block size and 128 in-flight requests per I/O queue, summing up to 256 pages for potential DMAs. The guest storage driver pre-maps 302 pages at boot time. Therefore, up to 558 pages may be mapped for DMA simultaneously, at any time. Obviously, coIOMMU precisely captures such temporal locality and constantly pins 558 pages in our test.

GPU - There is no recognizable difference between the line of *PT-N* and the bottom four lines, in OpenArena. The reason is

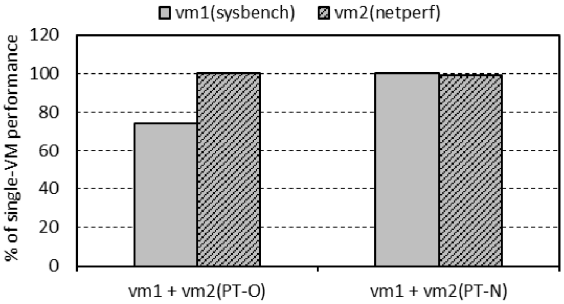
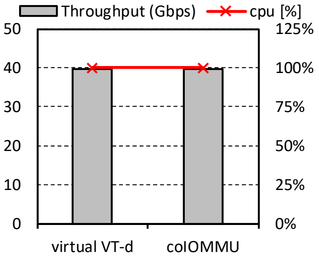


Figure 6: The impact of memory overcommitment: static pinning (*PT-O*) vs. fine-grained pinning (*PT-N*)



Spent Cycles

	virtual VT-d	coIOMMU	ratio
Create VM	7554ms	407ms	18x
Assign NIC	183ms	2ms	91x
Deassign NIC	815ms	2ms	407x

Pinned Pages

	virtual VT-d	coIOMMU	ratio
Before DPDK	8388608	548	15307x
In DPDK	186368	186368	1x
After DPDK	8838608	548	15307x

Figure 7: Running DPDK with virtual VT-d and coIOMMU

simple, as explained in §5.1, that OpenArena maps most of its DMA pages (~240MB) one-off at launch time and then unmaps them only at exit. In such circumstance, smart pinning and lazy unpinning have no effect at all. Therefore, all five fine-grained pinning modes pin the similar number of guest pages, with only static pinning staying in the top.

5.3. Memory Overcommitment

Overcommitment allows the aggregated size of all VMs to exceed the physical memory, thus improving memory utilization. We explore this configuration in both coIOMMU (*PT-N*) and the virtual VT-d (*PT-O*), to demonstrate the value of fine-grained pinning.

We create two VMs in the test machine with 64GB physical memory. VM1 has no assigned device and is configured with 32GB memory. It runs sysbench to randomly access a 16GB memory region. On the other hand, VM2 is assigned with an Intel i40e NIC and is configured with 48GB memory. It runs Netperf to send packets through the assigned NIC. The total memory size of the two VMs (80GB) exceeds the physical memory limitation.

We compare the performance of running them together to that of running each alone, in Figure 6. With the virtual VT-d, Netperf sustains the single-VM performance while sysbench suffers 25% performance drop. The drop is caused by frequent page swaps due to insufficient host memory. There is only 8.8GB left after statically pinning 48GB memory for VM2. The situation gets worse with random errors reported in VM1, when increasing the memory intensity of sysbench. Conversely, both VMs achieve their desired performance with coIOMMU, with 49GB free memory available even when two benchmarks are both running.

5.4. Guest User Space Driver

The guest kernel may directly assign a device to its user space for improved performance. However, kernel bypassing imposes the risk of DMA attacks from the user space. In such case, the guest kernel typically turns on DMA remapping of vIOMMU when the device is being assigned to the user space and then turns off remapping after the device is assigned backed to the kernel. In such circumstance, coIOMMU can help the hypervisor maintain fine-grained pinning reliably, while state-of-the-art vIOMMUs suffer from increased overhead by switching back and forth between static pinning and fine-grained pinning. We demonstrate such an example using DPDK pktgen, which offloads TCP packet processing from the guest user space to the assigned NIC. We run DPDK with coIOMMU and with the virtual VT-d respectively and show the comparison in Figure 7.

coIOMMU dramatically reduces the latency in several stages, compared with the virtual VT-d: (1) 18x reduction when the VM is created (407ms vs. 7554ms); (2) 91x reduction when the guest kernel assigns the NIC to user space DPDK (2ms vs. 183ms); and (3) 407x reduction when the NIC is assigned back to the guest kernel (2ms vs. 815ms). The cost of the emulated VT-d is mostly caused by pinning or unpinning the entire guest memory when switching to or away from static pinning. The VM creation phase suffers most because every guest page needs to be allocated and cleared in static pinning. In the meantime, coIOMMU pins no more than 186K pages, while the virtual VT-d pins many more pages varying between 186K and 8M.

5.5. DMA Temporal Locality

Good temporal locality on DMA buffers is crucial for high performance I/O processing, both in virtualization and on bare metal. Commercial OSes are optimized toward this goal, as observed in our evaluation and also reported by previous studies [7, 44, 51]. On the other hand, Markuze et al. [30] observes that many pages may be used to hold DMA buffers, over time, in stock Linux. Hence, we studied the DMA temporal locality of the networking stack in a similar configuration, by running 16 Netperf TX instances for 15 minutes, shown in Figure 8. We also run a Linux 'dd' command alongside Netperf, reading the raw virtual disk into /dev/zero. The 'dd' command constantly causes ~20K page cache misses per second, leading to ~20K new page allocated and heavily contending with the networking stack. The experiment is conducted in *PT-N* mode, i.e. under the passthrough policy.

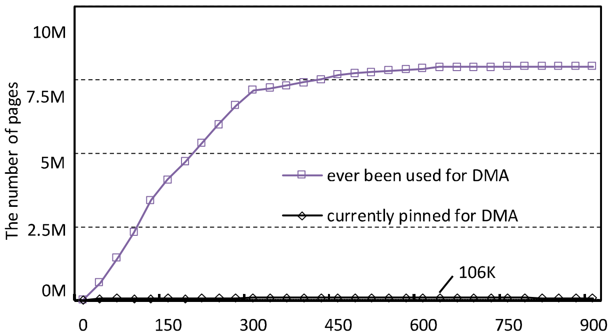


Figure 8: DMA temporal locality when running Netperf with 'dd'

Our data echoes the previous finding [30] – almost the entire guest memory (~7.9M pages, 98.7% of total memory) has ever been used for sending packets, over time. However, the number of pinned pages almost stays flat when coIOMMU is enabled. The peak number is ~106K (424MB), 2.4x of that when running Netperf TX alone and just 1.3% of the total guest memory. The result implies that the DMA locality in a short period is still good in such stress case, allowing the hypervisor to intelligently pin the guest pages with coIOMMU.

5.6. Mixed Workloads

We run Netperf TX and fio together to check how coIOMMU performs in mixed I/O workloads. The tested VM is configured with 16 vCPUs and 32GB memory as previous tests. It is directly assigned two devices: a XL710 NIC and a 760P SSD. We launch 16 netperf instances and 16 fio threads simultaneously in the VM, with each vCPU holding one netperf instance and one fio thread. Here we just compare *PT-O* vs. *PT-N* under the passthrough policy, as the two modes can best demonstrate the coIOMMU benefits according to the baseline data.

The result is promising. First, there is no observable performance difference when comparing Netperf and fio to their baseline performance of running alone. The deviations are less than 1% and within the error bar. Second, the peak number of pinned pages in mixed workloads is 45200 (176.5MB), close to the sum of pinned pages of running Netperf (174MB) and fio (2.2MB) alone. This result proves that coIOMMU can effectively reduce the memory footprint with negligible overhead, even when running mixed direct I/O usages together.

6. Conclusions and Future Work

Established vIOMMUs cannot reliably eliminate static pinning in direct I/O, due to the emulation cost of their DMA remapping interfaces. We instead propose coIOMMU, a new vIOMMU architecture for efficient memory management. coIOMMU introduces a cooperative DMA buffer tracking mechanism for fine-grained pinning, orthogonal to the costly DMA remapping interface. The new mechanism uses a shared DMA tracking table (DTT) for hypervisor and guest to exchange the DMA buffer information, without incurring excessive notifications from the guest, due to smart pinning and lazy unpinning. We demonstrate that coIOMMU not only dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost, but also sustains the desired security as required in specific protection policies. Last but not the least, although we implement coIOMMU by extending an emulated vIOMMU - the virtual Intel VT-d, this design can be easily ported to other vIOMMUs.

As for future work, we will focus on several areas. First, new IOMMU trends [53, 54] begin to support two-level address translations, allowing the guest to skip certain virtual IOTLB invalidations for improved performance. coIOMMU should provide efficient DMA buffer tracking in two-level translation and maintain its performance benefit. Second, some devices (e.g. GPUs) partially support DMA page faults, e.g. only for selective pages such as those used by applications. We want to study a hybrid approach for fine-grained pinning, by leveraging DMA page faults for faultable pages and using coIOMMU for other non-faultable pages. Last, kernel bypassing usually needs to pre-map a big trunk of memory for the application to manage. We want to extend the coIOMMU concept from the boundary between hypervisor and guest to the boundary between kernel space and user space, to enable finer-grained memory management in such usage.

References

[1] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian and Haibing Guan. High Performance Network Virtualization with SR-IOV. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2010. https://doi.org/10.1109/HPCA.2010.5416637.

[2] PCI-SIG. Address Translation Services Revision 1.1. http://www.pcisig.com/specifications/iov/ats/, 2009.

[3] Intel Corporation. Intel® Virtualization Technology for Directed I/O. Architecture specification, rev. 3.1. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Jun 2019.

[4] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, 2015. https://doi.org/10.1109/SP.2015.43.

[5] Khronos. The OpenCL Specification, rev 2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf, July 2015.

[6] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 449–466, 2016. https://doi.org/10.1145/3093337.3037710.

[7] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 18:1–18:12, 2010. https://doi.org/10.1145/1815695.1815718.

[8] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011. https://www.usenix.org/legacy/events/atc11/tech/final_files/Amit.pdf.

[9] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. In *IEEE Transactions on Computers*, volume 68, issue 4, 2019. https://doi.org/10.1109/TC.2018.2879080.

[10] Avi Kivity, Yaniv Kamay, and Dor Laor. kvm: the Linux Virtual

Machine Monitor. In *Ottawa Linux Symposium (OLS)*, pages 225-230, 2007. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[11] Chris Schlaeger. AWS EC2 Virtualization: Introducing Nitro. In *AWS Summit*, 2018. http://aws-de-media.s3.amazonaws.com/images/AWS_Summit_2018/June7/Alexandria/Introducing-Nitro.pdf.

[12] Alibaba Corporation. Ali cloud elastic bare metal server – Shenlong architecture (X-Dragon) secret. http://www.programmersought.com/article/7752222651/.

[13] PCI-SIG. Single root I/O virtualization and sharing 1.0 specification. https://pcisig.com/specifications/iov/single_root/, Sep 2007.

[14] AMD Corporation. AMD IOMMU architecture specification, rev 3.00. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf, De c 2016

[15] Christopher Clark, Keir Fraser, Seven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, volume 2, pages 273-286, 2005. https://www.usenix.org/legacy/event/nsdi05/tech/full_papers/clark/clark.pdf.

[16] ARM Corporation. ARM System Memory Management Unit Architec ture Specification, rev 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, 2016

[17] Intel Corporation. Intel Scalable I/O Virtualization Technical Specification, rev 1.0. https://software.intel.com/en-us/download/intel-scalable-io-virtualization-technical-specification, Jun 2018

[18] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-9-20.pdf.

[19] Amazon Corporation. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2019

[20] Xin Xu, Bhavesh Davda. SRVM: Hypervisor Support for Live Migration with Passthrough SR-IOV Network Devices. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 65-77, 2016. https://dl.acm.org/citation.cfm?doid=2892242.2892256.

[21] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-Metal Performance for Virtual Machines with Exitless Interrupts. In *Communications of ACM*, volume 59, issue 1, pages 108-116, 2016. https://doi.org/10.1145/2845648.

[22] Alibaba Corporation. Elastic Compute Service Instance Type Families. https://www.alibabacloud.com/help/doc-detail/25378.htm, Jul 2019.

[23] Eric Auger. vIOMMU/ARM: full emulation and virtio-iommu approaches. In *KVM Forum*, 2017. http://events17.linuxfoundation.org/sites/events/files/slides/viommu_arm.pdf.

[24] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-Free IOMMU Protection for Networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 301–315, 2018. https://doi.org/10.1145/3173162.3173175.

[25] James E.J. Bottomley. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt. Linux kernel documentation.

[26] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live Migration with Pass-through Device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261-268, 2008. https://landley.net/kdocs/ols/2008/ols2008v2-pages-261-267.pdf.

[27] The HSA Foundation. http://www.hsafoundation.com/.

[28] Asim Kadav and Michael M. Swift. Live Migration of Direct-Access Devices. In *ACM SIGOPS Operating System Review (OSR),* volume 43, issue 3, pages 95-104, 2009. http://pages.cs.wisc.edu/~swift/papers/shadow-migrate-osr.pdf.

[29] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 423-436, 2010. https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Ben-Yehuda.pdf.

[30] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–262, 2016. https://doi.org/10.1145/2872362.2872379.

[31] Ardalan Amri Sani, Kevin Boos, Shaoqu Qin, and Lin Zhong. I/O Paravirtualizatoin at the Device File Boundary. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. http://doi.org/10.1145/2541940.2541943.

[32] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org.

[33] AF_XDP. https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html. Linux networking documentation.

[34] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2003. https://suif.stanford.edu/papers/vmi-ndss03.pdf.

[35] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, Zhijiao Zhang. CompSC: Live Migration with Pass-through Devices. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 109-120, 2012. https://doi.org/10.1145/3139645.3139649.

[36] Yuval Yarom and Katrina Falkner. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX conference on Security Symposium (SEC)*, pages 719–732, 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf.

[37] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 1-15, 2015. https://doi.org/10.1145/2731186.2731189.

[38] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU Scalably. In *USENIX Annual Technical Conference (ATC)*, pages 549–562, 2011. https://www.usenix.org/system/files/conference/atc15/atc15-paper-peleg.pdf.

[39] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 6, pages 2-2, 2004. https://www.usenix.org/legacy/events/osdi04/tech/full_papers/levasseur/levasseur.pdf.

[40] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016. https://doi.org/10.1109/ISPASS.2016.7482091.

[41] Nikolay Sakharnykh. Everything You Need to Know About Unified Memory. In *NVIDIA's GPU Technology Conference (GTC)*, 2018. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

[42] Intel Corporation. Intel Open Source HD Graphics and Intel® Iris® Plus graphics Programmer's Reference Manual, page 139, 2017. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol05-memory_views.pdf.

[43] Vinod Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007.

[44] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *USENIX Annual Technical Conference (ATC)*, 2008. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann_html/.

[45] Alex Williamson. VFIO: A user's perspective. In *KVM Forum*, 2012. http://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf.

[46] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security Implications of Memory Deduplication in a Virtualized Environment. In

Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2013. https://www.eecis.udel.edu/~hnw/paper/memdedup.pdf.

[47] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient Intra-Operating System Protection Against Harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 29-44, 2015. https://www.usenix.org/system/files/conference/fast15/fast15-paper-malka.pdf.

[48] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)*, Pages 3-3, 2006. https://www.usenix.org/legacy/event/usenix06/tech/full_papers/liu/liu_html/usenix06.html.

[49] Himanshu Raj and Karsten Schwan. High Performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC)*, pages 189-188, 2007. https://doi.org/10.1145/1272366.1272390.

[50] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10Gbps Using Safe and Transparent Network Interface Virtualization. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2009. https://www.cs.rice.edu/~rixner/publication/ram-09/.

[51] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. IOMMU: strategies for mitigating the IOTLB bottleneck. In *Proceedings of International Conference on Computer Architecture (ISCA)*, pages 256-274, 2010. https://doi.org/10.1007/978-3-642-24322-6_22.

[52] Hongwei Tang, Qiang Li, Shengzhong Feng, Xiaofang Zhao, and Yan Jin. IOMMU Para-Virtualization for Efficient and Secure DMA in Virtual Machines. *In KSII Transactions on Internet and Information Systems*, vol. 10, no. 12, pp. 5938-5963, 2016. DOI: 10.3837/tiis.2016.12.014.

[53] Eric Auger. SMMUv3 Nested Stage Setup. https://lkml.org/lkml/2019/3/17/124.

[54] Baolu Lu. Use 1st-level for IOVA translation. https://lwn.net/Articles/807079/

[55] Jun Nakajima. Enhancing KVM for Guest Protection and Security. In *KVM Forum*, 2019. https://static.sched.com/hosted_files/kvmforum2019/23/nakajima-enhancing-kvm-for-guest-protection.pdf/.

[56] AMD. Secure Encrypted Virtualization. https://developer.amd.com/sev/.

[57] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2002. https://doi.org/10.1145/844128.844146

[58] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, 2005. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

[59] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. https://dl.acm.org/doi/pdf/10.1145/2775054.2694355/.

[60] Peter Xu. Device Assignment with Nested Guest and DPDK. In *KVM Forum*, 2017. https://www.linux-kvm.org/images/a/a6/KVM_Forum_2018_viommu_vfio.pdf.

[61] Baolu Lu. IOMMU: Bounce Page for Untrusted Devices. https://lwn.net/Articles/794595/.

[62] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Network and Distributed System Security (NDSS) Symposium*, 2019. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05A-1_Markettos_paper.pdf.

[63] Rick A. Jones. A network performance benchmark (revision 2.0). Technique report, Hewlett Packard, 1995. http://www.netperf.org/netperf/training/Netperf.html

[64] Nginx. https://www.nginx.com/.

[65] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004. https://memcached.org/.

[66] Fio. https://fio.readthedocs.io/en/latest/fio_doc.html.

[67] OpenArena. https://en.wikipedia.org/wiki/OpenArena.

[68] Sysbench. https://wiki.gentoo.org/wiki/Sysbench.

[69] Apachebench. http://en.wikipedia.org/wiki/ApacheBench.

[70] Brian Aker. Memslap – load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html.

BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning

Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan[†], Yang Liu[‡]
HKUST, [†]University of Nevada, Reno, [‡]WeBank
{czhangbn, slida, jxiaaf, weiwa}@cse.ust.hk, fyan@unr.edu, yangliu@webank.com

Abstract

Cross-silo federated learning (FL) enables organizations (e.g., financial or medical) to collaboratively train a machine learning model by aggregating local gradient updates from each client without sharing privacy-sensitive data. To ensure no update is revealed during aggregation, industrial FL frameworks allow clients to mask local gradient updates using *additively homomorphic encryption* (HE). However, this results in significant cost in *computation* and *communication*. In our characterization, HE operations dominate the training time, while inflating the data transfer amount by two orders of magnitude. In this paper, we present BatchCrypt, a system solution for cross-silo FL that substantially reduces the encryption and communication overhead caused by HE. Instead of encrypting individual gradients with full precision, we *encode a batch of quantized gradients* into a long integer and encrypt it in one go. To allow *gradient-wise aggregation* to be performed on *ciphertexts* of the encoded batches, we develop new quantization and encoding schemes along with a novel gradient clipping technique. We implemented BatchCrypt as a plug-in module in FATE, an industrial cross-silo FL framework. Evaluations with EC2 clients in geo-distributed datacenters show that BatchCrypt achieves 23×-93× training speedup while reducing the communication overhead by 66×-101×. The accuracy loss due to quantization errors is less than 1%.

1 Introduction

Building high-quality machine learning (ML) models requires collecting a massive amount of training data from diverse sources. However, in many industries, data is dispersed and locked in multiple organizations (e.g., banks, hospitals, and institutes), where data sharing is strictly forbidden due to the growing concerns about data privacy and confidentiality as well as violating the government regulations [12, 17, 45]. Cross-silo federated learning (FL) [27, 61] offers an appealing solution to break "data silos" among organizations, where participating clients *collaboratively learn* a global model by uploading their *local gradient updates* to a central server for aggregation, without sharing privacy-sensitive data.

To ensure that no client reveals its update during aggregation, many approaches have been proposed [9, 37, 47, 48, 52]. Among them *additively homomorphic encryption* (HE), notably the Paillier crytosystem [46], is particularly attractive in the cross-silo setting [37, 48, 61], as it provides a strong privacy guarantee at no expense of learning accuracy loss (§2). With HE, gradient aggregation can be performed on *ciphertexts* without decrypting them in advance. HE has been adopted in many cross-silo FL applications [13, 23, 37, 38, 44], and can be easily plugged into the existing FL frameworks to augment the popular parameter server architecture [33]. Before the training begins, an HE key-pair is synchronized across all clients through a secure channel. During training, each client encrypts its gradient updates using the public key and uploads the ciphertexts to a central server. The server aggregates the encrypted gradients from all clients and dispatches the result to each of them. A client decrypts the aggregated gradients using the private key, updates its local model, and proceeds to the next iteration. As clients only upload the encrypted updates, no information can be learned by the server or an external party during data transfer and aggregation.

Although HE provides a strong privacy guarantee for cross-silo FL, it performs complex cryptographic operations (e.g., modular multiplications and exponentiations) that are extremely expensive to compute. Our testbed characterization (§3) shows that more than 80% of the training iteration time is spent on encryption/decryption. To make matters worse, encryption yields substantially larger ciphertexts, inflating the amount of data transfer by over 150× than plaintext learning. The significant overhead of HE in *encryption* and *communication* has become a major roadblock to facilitating cross-silo FL. According to our contacts at WeBank [57], most of their FL applications cannot afford to use the encrypted gradients and are limited to scenarios with less stringent privacy requirements (e.g., FL across departments or trustworthy partners).

In this paper, we tackle the encryption and communication bottlenecks created by HE with a simple *batch encryption* technique. That is, a client first *quantizes* its gradient values into low-bit integer representations. It then *encodes* a batch of quantized values to a long integer and encrypts it in one go.

Compared with encrypting individual gradient values of full precision, batch encryption significantly reduces the encryption overhead and data transfer amount. Although this idea has been briefly mentioned in the previous work [37,48], the treatment is rather informal without a viable implementation. In fact, to enable batch encryption in cross-silo FL, there are two key technical challenges that must be addressed, which, to our knowledge, remains open.

First, a feasible batch encryption scheme should allow us to directly sum up the ciphertexts of two batches, and the result, when decrypted, matches that of performing *gradient-wise aggregation* on the two batches in the clear. We show that although it is viable to tweak the generic quantization scheme to meet such need, it has many limitations as it is not designed for aggregation. Instead, we design a customized quantization scheme that quantizes gradient values to *signed integers* uniformly distributed in a *symmetric* range. Moreover, to support gradient-wise aggregation in a simple additive form, and that the addition does not cause overflow to corrupt the encoded gradients, we develop a new batch encoding scheme that adopts *two's compliment representation* with *two sign bits* for quantized values. We also use *padding* and *advance scaling* to avoid overflow in addition. All these techniques allow gradient aggregation to be performed on ciphertexts of the encoded batches, without decryption first.

Second, as gradients values are *unbounded*, they need to be *clipped* before quantization, which critically determines the learning performance [5,41]. However, it remains unclear how to choose the clipping thresholds in the cross-silo setting. We propose an efficient analytical model dACIQ by extending ACIQ [5], a state-of-the-art clipping technique for ML over centralized data, to cross-silo FL over decentralized data. dACIQ allows us to choose optimal clipping thresholds with the minimum cumulative error.

We have implemented our solution BatchCrypt in FATE [18], a secure computing framework released by WeBank [57] to facilitate FL among organizations. Our implementation can be easily extended to support other optimization schemes for distributed ML such as local-update SGD [22, 35, 56], model averaging [40], and relaxed synchronization [24, 34, 62], all of which can benefit from BatchCrypt when applied to cross-silo FL. We evaluate BatchCrypt with nine participating clients geo-distributed in five AWS EC2 datacenters across three continents. These clients collaboratively learn three ML models of various sizes: a 3-layer fully-connected neural network with FMNIST dataset [60], AlexNet [32] with CIFAR10 dataset [31], and a text-generative LSTM model [25] with Shakespeare dataset [55]. Compared with the stock implementation of FATE, BatchCrypt accelerates the training of the three models by 23×, 71×, and 93×, respectively, where more salient speedup can be achieved for more complex models. In the meantime, the communication overhead is reduced by 66×, 71×, and 101×, respectively. The significant benefits of BatchCrypt come at no cost of model quality, with a negligible accuracy loss less than 1%. BatchCrypt[1] offers the first efficient implementation that enables HE in a cross-silo FL framework with low encryption and communication cost.

2 Background and Related Work

In this section, we highlight the stringent privacy requirements posed by cross-silo federated learning. We survey existing techniques for meeting these requirements.

2.1 Cross-Silo Federated Learning

According to a recent survey [27], federated learning (FL) is a scenario where multiple clients collaboratively train a machine learning (ML) model with the help of a central server; each client transfers local updates to the server for immediate aggregation, without having its raw data leaving the local storage. Depending on the application scenarios, federated learning can be broadly categorized into *cross-device* FL and *cross-silo* FL. In the cross-device setting, the clients are a large number of mobile or IoT devices with limited computing power and unreliable communications [27,30,39]. In contrast, the clients in the cross-silo setting are a small number of organizations (e.g., financial and medical) with reliable communications and abundant computing resources in datacenters [27,61]. We focus on cross-silo FL in this paper.

Compared with the cross-device setting, cross-silo FL has significantly more stringent requirements on privacy and learning performance [27,61]. *First*, the final trained model should be *exclusively released* to those participating organizations—no external party, including the central server, can have access to the trained model. *Second*, the strong privacy guarantee should not be achieved at a cost of learning accuracy. *Third*, as an emerging paradigm, cross-silo FL is undergoing fast innovations in both algorithms and systems. A desirable privacy solution should impose *minimum constraints* on the underlying system architecture, training mode (e.g., synchronous and asynchronous), and learning algorithms.

2.2 Privacy Solutions in Federated Learning

Many strategies have been proposed to protect the privacy of clients for federated learning. We briefly examine these solutions and comment on their suitability to cross-silo FL.

Secure Multi-Party Computation (MPC) allows multiple parties to collaboratively compute an agreed-upon function with private data in a way that each party knows nothing except its input and output (i.e., zero-knowledge guarantee). MPC utilizes carefully designed computation and synchronization protocols between clients. Such protocols have strong privacy guarantees, but are difficult to implement efficiently

[1] BatchCrypt is open-sourced and can be found at https://github.com/marcoszh/BatchCrypt

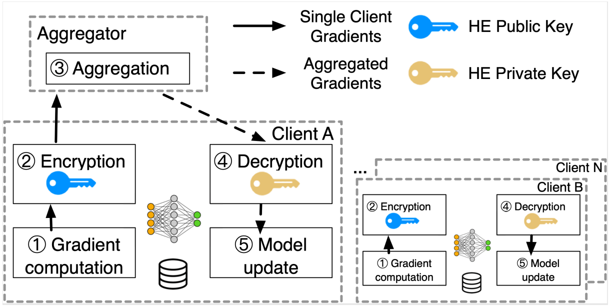


Figure 1: The architecture of cross-silo FL system, where HE is implemented as a pluggable module on the clients.

in a geo-distributed scenario like cross-silo FL [61]. Developers have to carefully engineer the ML algorithms and divide the computation among parties to fit the MPC paradigm, which may lower the privacy guarantees for better performance [16, 42, 43].

Differential Privacy (DP) is another common tool that can be combined with model averaging and SGD to facilitate secure FL [47, 52]. It ensures the privacy of each individual sample in the dataset by injecting noises. A recent work proposes to employ *selective parameter update* [52] atop differential privacy to navigate the tradeoff between data privacy and learning accuracy. Although DP can be efficiently implemented, it exposes plain gradients to the central server during aggregation. Later study shows that one can easily recover the information from gradients [48]. While such privacy breach and the potential accuracy drop might be tolerable for mobile users in cross-device FL, they raise significant concerns for participating organizations in cross-silo FL.

Secure Aggregation [9] is proposed recently to ensure that the server learns no individual updates from any clients but the *aggregated updates* only. While secure aggregation has been successfully deployed in cross-device FL, it falls short in cross-silo FL for two reasons. First, it allows the central server to see the aggregated gradients, based on which the information about the trained model can be learned by an external entity (e.g., public cloud running the central server). Second, in each iteration, clients must synchronize secret keys and *zero-sum masks*, imposing a strong requirement of *synchronous training*.

Homomorphic Encryption (HE) allows certain computation (e.g., addition) to be performed directly on ciphertexts, without decrypting them first. Many recent works [13, 37, 38, 48] advocate the use of additively HE schemes, notably Paillier [46], as the primary means of privacy guarantee in cross-silo FL: each client transfers the encrypted local updates to the server for direct aggregation; the result is then sent back to each client for local decryption. HE meets the three requirements of cross-silo FL. *First*, it protects the trained model from being learned by any external parties including the server as update aggregation is performed on ciphertexts. *Second*, it incurs no learning accuracy loss, as no noise is added to the model updates during the encryption/decryption process. *Third*, HE directly applies to the existing learning systems, requiring no modifications other than encrypting/decrypting updates. It hence imposes no constraints to the synchronization schemes and the learning algorithms. However, as we shall show in §3, HE introduces significant overhead to computation and communication.

Summary To summarize, each of these privacy-preserving techniques has its pros and cons. MPC is able to provide strong privacy guarantees, but requires expert efforts to re-engineer existing ML algorithms. DP can be adopted easily and efficiently, but has the downside of weaker privacy guarantee and potential accuracy loss. Secure aggregation is an effective way to facilitate large-scale cross-device FL, but may not be suitable for cross-silo FL as it exposes the aggregated results to third parties and incurs high synchronization cost. HE can be easily adopted to provide strong privacy guarantees without algorithm modifications or accuracy loss. However, the high computation and communication overheads make it impractical for production deployment at the moment.

2.3 Cross-Silo FL Platform with HE

Fig. 1 depicts a typical cross-silo FL system [27, 37, 61], where HE is implemented as a pluggable module on the clients. The *aggregator* is the server which coordinates the *clients* and aggregates their encrypted gradients. Note that in this work, we assume the aggregator is *honest-but-curious*, a common threat model used in the existing FL literature [9, 38, 52]. The communications between all parties (the clients and the aggregator) are secured by cryptographic protocols such as SSL/TLS, so that no third party can learn the messages being transferred. Before the training starts, the aggregator randomly selects a client as the leader who generates an HE key-pair and synchronizes it to all the other clients. The leader also initializes the ML model and sends the model weights to all the other clients. Upon receiving the HE key-pair and the initial weights, the clients start training. In an iteration, each client computes the local gradient updates (①), encrypts them with the public key (②), and transfers the results to the aggregator. The aggregator waits until the updates from all the clients are received. It then adds them up and dispatches the results to all clients (③). A client then decrypts the aggregated gradients (④) and uses it to update the local model (⑤).

This architecture design follows the classic distributed SGD pattern. So the existing theories and optimizations including flexible synchronization [24, 34, 62] and local update SGD [22, 35, 56] naturally apply. Moreover, as model updating is performed on the client's side using the plaintext gradient aggregation, we can adopt state-of-the-art adaptive optimizers such as Adam [28] for faster convergence—a huge advantage over the existing proposal [48] that applies encrypted gradients directly on the encrypted global model in the server.

3 Characterizing Performance Bottlenecks

In this section, we characterize the performance of cross-silo FL with three real applications driven by deep learning models in a geo-distributed setting. We show that encryption and communication come as two prohibitive bottlenecks that impede the adoption of FL among organizations. We survey possible solutions in the literature and discuss their inefficiency. To our knowledge, we are the first to present a comprehensive characterization for cross-silo FL in a realistic setting.

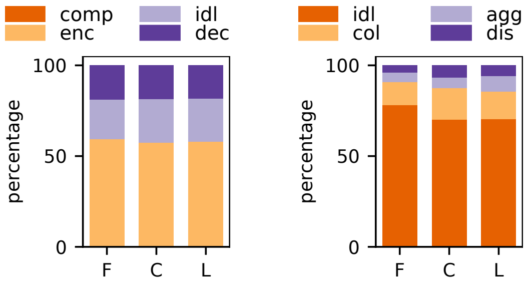
3.1 Characterization Results

Cross-silo FL is usually performed in multiple *geo-distributed* datacenters of participating organizations [27, 61]. Our characterization is carried out in a similar scenario where nine EC2 clients in five geo-distributed datacenters collaboratively training three ML models of various sizes, including FMNIST, CIFAR, and LSTM (Table 3). Unless otherwise specified, we configure *synchronous training*, where no client can proceed to the next iteration until the (encrypted) updates from all clients have been aggregated. We defer the detailed description of the cluster setup and the ML models to §6.1.

We base our study in FATE (Federated AI Technology Enabler) [18], a secure compute framework developed by WeBank [57] to drive its FL applications with the other industry partners. To our knowledge, FATE is the only open-source cross-silo FL framework deployed in production environments. FATE has a built-in support to the Pailler cryptosystem [46] (key size set to 2048 bits by default), arguably the most popular additively HE scheme [50]. Our results also apply to the other partially HE cryptosystems.

Encryption and Communication Overhead We start our characterization by comparing two FL scenarios, with and without HE. We find that the use of HE results in *exceedingly long training time* with *dramatically increased data transfer*. More specifically, when HE is enabled, we measured the average training iteration time 211.9s, 2725.7s, and 8777.7s for FMNIST, CIFAR, and LSTM, respectively. Compared with directly transferring the plaintext updates, the iteration time is extended by 96×, 135×, and 154×, respectively. In the meantime, when HE is (not) in use, we measured 1.1GB (6.98MB), 13.1GB (85.89MB), and 44.1GB (275.93MB) data transfer between clients and aggregator in one iteration on average for FMNIST, CIFAR, and LSTM, respectively. To sum up, the use of HE increases both the training time and the network footprint by two orders of magnitude. Such performance overhead becomes even more significant for complex models with a large number of weights (e.g., LSTM).

Deep Dive To understand the sources of the significant overhead caused by HE, we examine the training process of the three models in detail, where we sample an iteration and depict in Fig. 2 the breakdown of the iteration time spent on different operations on the client's side (left) and on the aggregator's side (right), respectively.



(a) Client: **comp**ute, **enc**rypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

Figure 2: Iteration time breakdowns of **F**MNIST, **C**IFAR, and **L**STM for a client and the aggregator.

As illustrated in Fig. 2a, on the client's side, HE-related operations dominate the training time in all three applications. In particular, a client spent around 60% of the iteration time on gradient encryption (yellow), 20% on decryption (dark purple), and another 20% on data transfer and idle waiting for the gradient aggregation to be returned[2] (light purple). In comparison, the time spent on the actual work for computing the gradients becomes *negligible* ($< 0.5\%$).

When it comes to the aggregator (Fig. 2b), most of the time ($> 70\%$) is wasted on idle waiting for a client to send in the encrypted gradients (orange). Collecting the gradients from all clients (yellow) and dispatching the aggregated results to each party (dark purple) also take a significant amount of time, as clients are geo-distributed and may not start transferring (or receiving) at the same time. The actual computation time for gradient aggregation (light purple) only accounts for less than 10% of the iteration span. Our deep-dive profiling identifies encryption and decryption as the two dominant sources of the exceedingly long training time.

Why is HE So Expensive? In additively HE cryptosystems such as Paillier [46], encryption and decryption both involve multiple modular multiplications and exponentiation operations with a large exponent and modulus (usually longer than 512 bits) [50], making them extremely expensive to compute. Encryption also yields significantly larger ciphertexts, which, in turn, causes a huge communication overhead for data transfer. In additively HE schemes such as Paillier, a ciphertext takes roughly the same number of bits as the key size, irrespective of the plaintext size. As of 2019, the minimum secure key size for Paillier is 2048 [6], whilst a gradient is typically a 32-bit floating point. This already translates to 64× size inflation after encryption.

We further benchmark the computation overhead and the inflated ciphertexts of Paillier with varying key sizes. We use `python-paillier` [15] to encrypt and then decrypt 900K 32-bit floating points. Table 1 reports the results on

[2] Due to the synchronization barrier, a client needs to wait for all the other clients to finish transferring updates to the aggregator.

Table 1: Benchmarking Paillier HE with various key sizes.

Key size	Plaintext	Ciphertext	Encryption	Decryption
1024	6.87MB	287.64MB	216.87s	68.63s
2048	6.87MB	527.17MB	1152.98s	357.17s
3072	6.87MB	754.62MB	3111.14s	993.80s

a c5.4xlarge instance. As the key size increases (higher security), both the computation overhead and the size of ciphertexts grow linearly. Since Paillier can only encrypt integers, floating point values have to be scaled beforehand, and their exponents information contribute further to data inflation.

Summary The prohibitive computation and communication overhead caused by HE, if not properly addressed, would lead to two serious economic consequences. First, given the dominance of HE operations, accelerating model computation using high-end hardware devices (e.g., GPUs and TPUs) is no longer relevant—a huge waste of the massive infrastructure investments in clients' datacenters. Second, the overwhelming network traffics across geo-distributed datacenters incurs skyrocketing Internet data charges, making cross-silo FL economically unviable. In fact, in WeBank, production FL applications may choose to turn off HE if the security requirement is not so strict.

3.2 Potential Solutions and Their Inefficiency

Hardware-Accelerated HE HE process can be accelerated using software or hardware solutions. However, typical HE cryptosystems including Paillier have limited interleaving independent operations, thus the potential speedup of a single HE operation is quite limited. In fact, it is reported that a specialized FPGA can only accelerate Paillier encryption by 3× [50]. Moreover, simply accelerating the encryption itself does not help reduce the communication overhead.

Reducing Communication Overhead As accelerating HE itself does not clear the barrier of adopting HE in FL, what if we reduce the amount of data to encrypt in the first place? Since data inflation is mainly caused by the mismatch between the lengths of plaintexts and ciphertexts, an intuitive idea would be *batching* as many gradients together as possible to form a *long* plaintext, so that the amount of encryption operations will reduce greatly. However, the challenge remains how to maintain HE's additive property after batching without modifying ML algorithms or hurting the learning accuracy.

While some prior works have explored the idea of joining multiple values together to reduce HE overhead, they give no viable implementation of batch encryption for cross-silo FL. [48] makes a false assumption that quantization is lossless, and uses adaptive optimizer Adam in its simulation even though its design does not support that. With only plain SGD available, [48] requires tedious learning rate scheduling tuning to achieve similar results of advanced optimizers [59]. The naive batching given in [37] cannot be correctly implemented as homomorphic additivity is not retained. In fact, none of these works have systematically studied the impact of batching. Gazelle [26] and SEAL [51] adopt the SIMD (single instruction multiple data) technique to speed up HE. However, such approach only applies to lattice-based HE schemes [11] and is restricted by their unique properties. For instance, it incurs dramatic computational complexity for lattice-based HE schemes to support more levels of multiplication [26]. Besides, these works only accelerate integer cryptographic operations. How to maintain the training accuracy in cross-silo FL context remains an open problem.

4 BatchCrypt

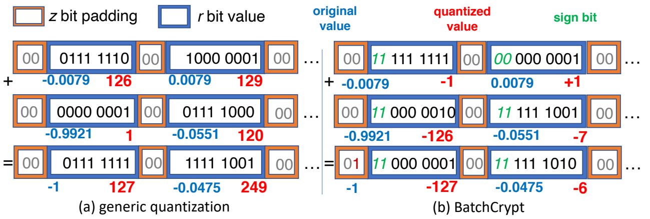
In this section, we describe our solution for gradient batching. We begin with the technical challenges. We first show that gradient quantization is required to enable batching. We then explain that generic quantization scheme lacks flexibility and efficiency to support general ML algorithms, which calls for an appropriately designed encoding and batching scheme; to prevent model quality degradation, an efficient clipping method is also needed. We name our solution BatchCrypt, a method that co-designs quantization, batch encoding, and analytical quantization modeling to boost computation speed and communication efficiency while preserving model quality in cross-silo FL with HE.

4.1 Why is HE Batching for FL a Problem?

On the surface, it seems straightforward to implement gradient batching. In fact, batching has been used to speed up queries over integers in a Paillier-secured database [19]. However, this technique only applies to *non-negative integers* [19]. In order to support floating numbers, the values have to be *reordered and grouped by their exponents* [19]. Such constraints are the key to preserving HE's additivity of batched ciphertexts—that is, the sum of two batched ciphertexts, once decrypted, should match the results of element-wise adding plaintext values in the two groups. Gazelle and SEAL [26, 51] employ SIMD technique to meet this requirement, but the approach is limited to lattice-based cryptosystems. We aspire to propose a universal batching method for all additively homomorphic cryptosystems.

Why Quantization is Needed? Gradients are *signed floating values* and must be ordered by their corresponding model weights, for which we cannot simply rearrange them by exponents. The only practical approach is to use integer representations of gradients in the batch, which requires quantization.

Existing Quantization Schemes ML algorithms are resilient to update noise and able to converge with gradients of limited precision [10]. Fig. 3a illustrates how generic gradient quantization scheme can be used in HE batching. Notably, since there is no bit-wise mapping between a ciphertext and its plaintext, permutation within ciphertexts is not allowed—only plain bit-by-bit addition between batched integers is available. Assume a gradient g in $[-1, 1]$ is quantized into an



(a) generic quantization

(b) BatchCrypt

Figure 3: An illustration of a generic quantization scheme and BatchCrypt. The latter preserves additivity during batching, with the sign bits highlighted within values.

8-bit unsigned integer. Let $[\cdot]$ denote the standard rounding function. The quantized value of g is

$$Q(g) = [255 * (g - min)/(max - min)],$$

where $max = 1$ and $min = -1$. Suppose n quantized gradients are summed up. The result, denoted by q_n, is dequantized as

$$Q^{-1}(q_n) = q_n * (max - min)/255 + n * min.$$

Referring to Fig. 3a, gradients of a client (floating numbers in blue) are first quantized and then batch joined into a large integer. To aggregate the gradients of two clients, we simply sum up the two batched integers, locate the added gradients at the same bit positions as in the two batches (8-bit integers in red), and dequantize them to obtain the aggregated results.

Such a generic quantization scheme, though simple to implement, does not support aggregation well and has many ***limitations*** when applied to batched gradient aggregation.

(1) It is restrictive. In order to dequantize the results, it must know how many values are aggregated. This poses extra barriers to flexible synchronization, where the number of updates is constantly changing, sometimes even unavailable.

(2) It overflows easily in aggregation. As values are quantized into positive integers, aggregating them is bound to overflow quickly as the sum grows larger. To prevent overflow, batched ciphertexts have to be decrypted after a few additions and encrypted again in prior work [48].

(3) It does not differentiate *positive* overflows from *negative*. Once overflow occurs, the computation has to restart. Should we be able to tell them apart, a saturated value could have been used instead of discarding the results.

4.2 HE Batching for Gradients

Unsatisfied with the generic quantization technique, we aspire to devise a batching solution tailored to gradient aggregation. Our scheme should have the following desirable properties: (1) it preserves the additivity of HE; (2) it is more resilient to overflows and can distinguish positive overflows from negative ones; (3) it is generally applicable to existing ML algorithms and optimization techniques; (4) it is flexible enough that one can dequantize values directly without additional information, such as the number of values aggregated.

Gradient Quantization Existing works use gradient compression techniques to reduce network traffic in distributed training [1,29,36,58]. These quantization methods are mainly used to compress values for transmission [58] or accelerate inference where only multiplication is needed [5]. However, they are not designed for gradient aggregation, and we cannot perform computations over the compressed gradients efficiently, making them inadequate for FL. We scrutinize the constraints posed by our design objectives, and summarize the stemed requirements for quantization as follows:

- **Signed Integers:** Gradients should be quantized into *signed* integers. In this way, positive and negative values can cancel each other out in gradient aggregation, making it less prone to overflowing than quantizing gradients into unsigned integers.
- **Symmetric Range:** To make values with opposite signs cancel each other out, the quantized range must be symmetrical. Violating this requirement may lead to an incorrect aggregation result. For example, if we map $[-1,1]$ to $[-128,127]$, then $-1+1$ would become $-128+127 = -1$ after quantization.
- **Uniform Quantization:** Literature shows that non-uniform quantization schemes have better compression rates as gradients have non-uniform distribution [1, 7]. However, we are unable to exploit the property as additions over quantized values are required.

BatchCrypt We now propose an efficient quantization scheme BatchCrypt that meets all the requirements above. Assume that we quantize a gradient in $[-\alpha, \alpha]$ into an r-bit integer. Instead of mapping the whole range all together, we *uniformly map* $[-\alpha, 0]$ and $[0, \alpha]$ to $[-(2^r - 1), 0]$ and $[0, 2^r - 1]$, respectively. Note that the value 0 ends up with two codes in our design. Prior work shows that 16-bit quantization ($r = 16$) is sufficient to achieve near lossless gradient quantization [21]. We will show in §6 that such a moderate quantization width is sufficient to enable efficient batching in FL setting.

With quantization figured out, the challenge remains how to encode the quantized values so that signed additively arithmetic is correctly enabled—once the batched long integer is encrypted, we cannot distinguish the sign bits from the value bits during aggregation. Inspired by how modern CPUs handle signed integer computations, we use *two's complement representation* in our encoding. By doing so, the sign bits can engage in the addition just like the value bits. We further use the *two sign bits* to differentiate between the positive and negative overflows. We illustrate an example of BatchCrypt in Fig. 3b. By adding the two batched long integers, BatchCrypt gets the correct aggregation results for $-1 + (-126)$ and $+1 + (-7)$, respectively.

BatchCrypt achieves our requirements by co-designing quantization and encoding: no additional information is needed to dequantize the aggregated results besides the batch itself; positive and negative values are able to offset each other; the signs of overflow can be identified. Compared

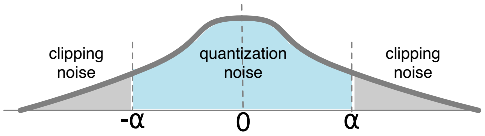


Figure 4: A typical layer gradient distribution. α is the clipping threshold.

with the batching methods in [26, 51], BatchCrypt's batching scheme is generally applicable to all additively HE cryptosystems' and fully HE cryptosystems' additive operations.

4.3 dACIQ: Analytical Clipping for FL

Our previous discussion has assumed gradients in a bounded range (§4.2). In practice, however, gradients may go unbounded and need to be *clipped* before quantization. Also, gradients from different layers have different distributions [58]. We thus need to quantize layers individually [1, 58]. Moreover, prior works show that gradients from the same layer have a bell-shaped distribution which is near Gaussian [2, 7, 53]. Such property can be exploited for efficient gradient compression [1, 58]. Finally, gradients require *stochastic rounding* during quantization [21, 36, 58], as it stochastically preserves diminishing information compared to *round-to-nearest*.

Layer-wise quantization and stochastic rounding can be easily applied, yet it remains unclear how to find the optimal clipping thresholds in the FL setting. As shown in Fig. 4, clipping is the process of saturating the outlaying gradients beyond a threshold α. If α is set too large, the quantization resolution becomes too low. On the other hand, if α gets too small, most of the range information from outlaying gradients has to be discarded.

In general, there are two ways to set the clipping threshold, *profiling-based* methods and *analytical modeling*. Profiling-based clipping selects a sample dataset to obtain a sample gradient distribution. Thresholds are then assessed with metrics such as KL divergence [41] and convergence rate [58]. However, such approach is impractical in FL for three reasons. First, finding a representative dataset in FL can be difficult, as clients usually have non-i.i.d. data, plus it breaks the data silo. Second, the gradient range narrows slowly as the training progresses [14], so clipping needs to be calibrated constantly, raising serious overhead concerns. Third, the profiling results are specific to the training models and datasets. Once the models or the datasets change, new profiling is needed. For both practicality and cost considerations, BatchCrypt instead adopts analytical modeling.

As shown in Fig. 4, the accumulated noise comes from two sources. *Quantization noise* refers to the error induced by rounding within the clipping range (the light blue area), while *clipping noise* refers to the saturated range beyond the clipping threshold (the gray area). To model the accumulated noise from both quantization and clipping, state-of-the-art clipping technique ACIQ [5] assumes that they follow a Gaussian distribution. However, ACIQ cannot be directly applied to BatchCrypt for two reasons. First, it employs a generic asymmetric quantization, which is not the case in BatchCrypt; second, in FL, gradients are not available at one place in plaintext to conduct distribution fitting.

We address these problems by extending ACIQ clipping to the distributed FL setting, which we call dACIQ. In particular, we adopt stochastic rounding with an r-bit quantization width. Assume that gradients follow Gaussian distribution $X \sim N(0, \sigma^2)$. Let q_i be the i-th quantization level. We compute the accumulated error in BatchCrypt as follows:

$$\begin{aligned} E[(X-Q(X))^2] &= \int_{-\infty}^{-\alpha} f(x)\cdot(x+\alpha)^2 dx + \int_{\alpha}^{\infty} f(x)\cdot(x-\alpha)^2 dx \\ &+ \sum_{i=0}^{2^r-3} \int_{q_i}^{q_{i+1}} f(x) \cdot [\, (x-q_i)^2 \cdot (\frac{q_{i+1}-x}{\triangle}) + (x-q_{i+1})^2 \cdot (\frac{x-q_i}{\triangle})\,] dx \\ &\approx \frac{\alpha^2+\sigma^2}{2} \cdot [1 - erf(\frac{\alpha}{\sqrt{2}\sigma})] - \frac{\alpha\cdot\sigma\cdot e^{-\frac{\alpha^2}{2\cdot\sigma^2}}}{\sqrt{2\pi}} + \frac{2\alpha^2\cdot(2^r-2)}{3\cdot 2^{3r}}, \end{aligned} \tag{1}$$

where the first and the second terms account for the clipping noise, and the third the rounding noise. As long as we know σ, we can then derive the optimal threshold α from Eq. (1). We omit the detailed derivations in the interest of space.

Gaussian Fitting Now that we have Eq. (1), we still need to figure out how to fit gradients into a Gaussian distribution in the FL setting. Traditionally, to fit Gaussian parameters μ and σ, Maximum Likelihood Estimation and Bayesian Inference can be used. They require information including the size of observation set, its sum, and its sum of squares. As an ML model may have up to millions of parameters, calculating these components as well as transferring them over Internet is prohibitively expensive. As a result, dACIQ adopts a simple, yet effective Gaussian fitting method proposed in [4]. The method only requires the size of observation set and its max and min, with the minimum computational and communication overhead. We later show that such light-weight fitting does not affect model accuracy in §6.

Advance Scaling With multiple clients in FL, it is essential to prevent overflows from happening. Thanks to clipping, the gradient range is predetermined before encryption. Let m be the number of clients. If m is available, we could employ *advance scaling* by setting the quantization range to m times of the clipping range, so that the sum of gradients from all clients will not overflow.

4.4 BatchCrypt: Putting It All Together

Putting it all together, we summarize the workflow of BatchCrypt in Algorithm 1.

Initialization The aggregator randomly selects one client as the leader. The leader client generates the HE key-pair and initializes the model weights. The key-pair and model weights are then synchronized with the other client workers.

Training After initialization, there is no differentiation between the leader and the other workers. Clients compute gra-

Algorithm 1 HE FL BatchCrypt

```
Aggregator:
function InitializE
    Issue InitializeLeader() to the randomly selected leader
    Issue InitializeOther() to the other clients
function StartsTraining
    for epoch $e = 0, 1, 2, ..., E$ do
        Issue WorkerStartsEpoch($e$) to all clients
        for all training batch $t = 0, 1, 2, \cdots, T$ do
            Collect gradients range and size
            Return clipping values $\alpha$ calculated by dACIQ
            Collect, sum up all $g_i^{(e,t)}$ into $g^{(e,t)}$, and dispatch it

Client Worker: $i = 1, 2, \ldots, m$
    – $r$: quantization bit width, $bs$: BatchCrypt batch size
function InitializeLeader
    Generate HE key-pair pub_key and pri_key
    Initialize the model to train w
    Send pub_key, pri_key, and w to other clients
function InitializeOther
    Receive HE key-pair pub_key and pri_key
    Receive the initial model weights w
function WorkerStartsEpoch($e$)
    for all training batch $t = 0, 1, 2, \cdots, T$ do
        Compute gradients $g_i^{(e,t)}$ based on $w$
        Send per-layer range and size of $g_i^{(e,t)}$ to aggregator
        Receive the layer-wise clipping values $\alpha$'s
        Clip $g_i^{(e,t)}$ with corresponding $\alpha$, quantize $g_i^{(e,t)}$ into $r$ bits, with
    quantization range setting to $m\alpha$                ▷ Advance scaling
        Batch $g_i^{(e,t)}$ with $bs$ layer by layer
        Encrypt batched $g_i^{(e,t)}$ with pri_key
        Send encrypted $g_i^{(e,t)}$ to aggregator
        Collect $g^{(e,t)}$ from aggregator, and decrypt with pub_key
        Apply decrypted $g^{(e,t)}$ to $w$
```

dients and send the per-layer gradient range and size to the aggregator. The aggregator estimates the Gaussian parameters first and then calculates the layer-wise clipping thresholds as described in § 4.3. Clients then quantize the gradients with range scaled by the number of clients, and encrypt the quantized values using BatchCrypt. Note that advanced scaling utilizing the number of clients is used to completely avoid overflowing. However, Algorithm 1 is still viable even without that information, as BatchCrypt supports overflow detection. The encrypted gradients are gathered at the aggregator and summed up before returning to the clients.

5 Implementation

We have implemented BatchCrypt atop FATE (v1.1) [18]. While we base our implementation on FATE, nothing precludes it from being extended to the other frameworks such as TensorFlow Federated [20] and PySyft [49].

Overview Our implementation follows the paradigm described in Algorithm 1, as most of the efforts are made on the client side. Fig. 5 gives an overview of the client architecture.

BatchCrypt consists of dACIQ, Quantizer, two's Compliments Codec, and Batch Manager. dACIQ is responsible for Gaussian fitting and clipping threshold calculation. Quantizer takes the thresholds and scales them to quantize the clipped values into signed integers. Quantizer also performs dequantization. Two's Compliments Codec translates between a quantized value's true form and two's compliment form with two sign bits. Given the large volume of data to encode, we adopt Numba to enable faster machine codes and massive parallelism. Finally, Batch Manager is in charge of batching and unbatching gradients in their two's compliment form, it remembers data's original shape before batching and restores it during unbatching. Batch Manager utilizes joblib to exploit computing resources by multiprocessing. FATE is used as an infrastructure to conduct FL, in which all the underlying ML computations are written with TensorFlow v1.14 optimized for our machines shipped with AWS DLAMI [3]. FATE adopts the open-sourced python-paillier as the Paillier HE implementation. We again employ joblib to parallel the operations here. FATE's Communication Manager conducts the SSL/TLS secured communication with gRPC. During our characterizations and evaluations, the CPUs are always fully utilized during Paillier operations and BatchCrypt process.

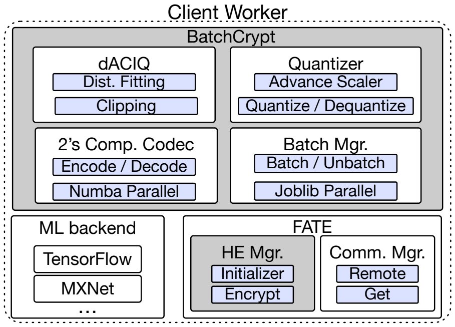


Figure 5: The architecture of a client worker in BatchCrypt.

Model Placement In the typical parameter server architecture, model weights are placed on the server side, while we purposely place weights on the worker side in BatchCrypt. Prior work [48] employs the traditional setup: clients encrypt the initialized weights with HE and send them to the aggregator first; the aggregator applies the received encrypted gradients to the weights encrypted with the same HE key. Such placement has two major drawbacks. First, keeping weights on the aggregator requires *re-encryption*. Since new gradients are constantly applied to weights, the model has to be sent back to the clients to decrypt and re-encrypt to avoid overflows from time to time, resulting in a huge overhead. Second, applying encrypted gradients prevents the use of sophisticated ML optimizers. State-of-the-art ML models are usually trained with adaptive optimizers [28] that scale the learning rates according to the gradient itself. By keeping the model weights on the client side, BatchCrypt can examine the aggregated plaintext gradients, enabling the use of advanced optimizers like Adam, whereas on the aggregator side, one can only adopt plain SGD.

Table 2: Network bandwidth (Mbit/sec) between aggregator and clients in different regions.

Region	Ore.	TYO.	N.VA.	LDN	HK
Uplink (Mbps)	9841	116	165	97	81
Downlink (Mbps)	9841	122	151	84	84

6 Evaluation

In this section, we evaluate the performance of BatchCrypt with real ML models trained in geo-distributed datacenters. We first examine the learning accuracy loss caused by our quantization scheme (§6.2). We then evaluate the computation and communication benefits BatchCrypt brings as well as how its performance compares to the ideal plaintext learning (§6.3). We then assess how BatchCrypt's speedup may change with various batch sizes (§6.4). Finally, we demonstrate the significant cost savings achieved by BatchCrypt (§6.5).

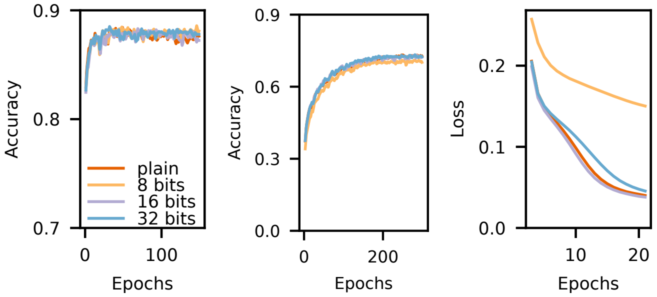
6.1 Methodology

Setting We consider a geo-distributed FL scenario where nine clients collaboratively train an ML model in five AWS EC2 datacenters located in Tokyo, Hong Kong, London, N. Virginia, and Oregon, respectively. We launched two compute-optimized `c5.4xlarge` instances (16 vCPUs and 32 GB memory) as two clients in each datacenter except that in Oregon, where we ran only one client. Note that we opt to not use GPU instances because computation is not a bottleneck. We ran one aggregator in the Oregon datacenter using a memory-optimized `r5.4xlarge` instance (16 vCPUs and 128 GB memory) in view of the large memory footprint incurred during aggregation. To better outline the network heterogeneity caused by geo-locations, we profiled the network bandwidth between the aggregator and the client instances. Our profiling results are summarized in Table 2. We adopt Pailler cryptosystem in our evaluation as it is widely adopted in FL [50], plus batching over it is not supported by Gazelle or SEAL [26, 51]. We expect our results also apply to other cryptosystems as BatchCrypt offers a generic solution.

Benchmarking Models As there is no standard benchmarking suites for cross-silo FL, we implemented three representative ML applications in FATE v1.1. Our first application is a 3-layer fully-connected neural network trained over FMNIST dataset [60], where we set the training batch size to 128 and adopt Adam optimizer. In the second application, we train AlexNet [32] using CIFAR10 dataset [31], with batch size 128 and RMSprop optimizer with 10^{-6} decay. The third application is an LSTM model [25] with Shakespeare dataset [55], where we set the batch size to 64 and adopt Adam optimizer. Other LSTM models that are easier to validate have significantly more weights. Training them to convergence is beyond our cloud budget. As summarized in Table 3, all three applications are backed by deep learning models of various sizes and cover common learning tasks such as image classification and text generation. For each application, we randomly partition its training dataset across nine clients. We configure synchronous training unless otherwise specified.

Table 3: Summary of models used in characterizations.

	FMNIST	**CIFAR**	**LSTM**
Network	3-layer FC	AlexNet [32]	LSTM [25]
Weights	101.77K	1.25M	4.02M
Dataset	FMNIST [60]	CIFAR10 [31]	Shakespeare [55]
Task	Image class.	Image class.	Text generation



(a) FMNIST test acc.

(b) CIFAR test acc.

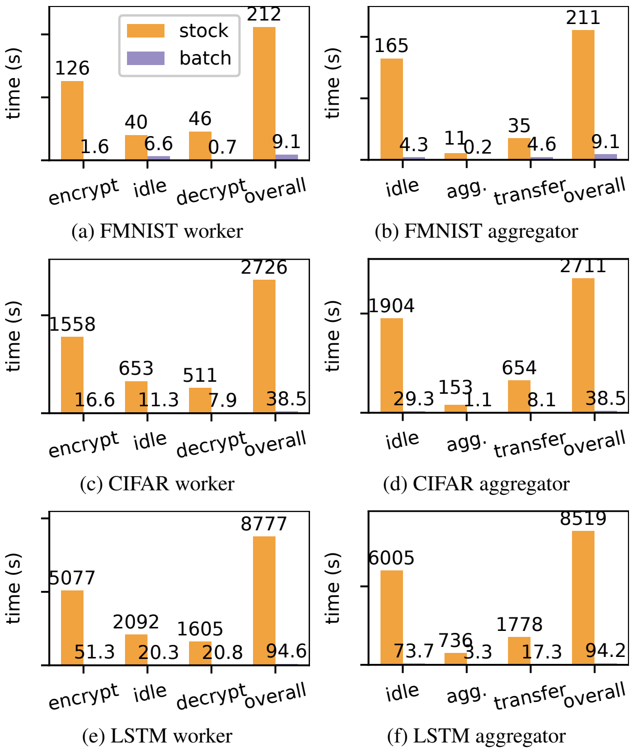
(c) LSTM train loss

Figure 6: The quality of trained model with different quantization bit widths in BatchCrypt.

6.2 Impact of BatchCrypt's Quantization

We first evaluate the impact of our quantization scheme, and see how quantization bit width could affect the model quality. We report the test accuracy for FMNIST and CIFAR workloads to see how BatchCrypt's quantization affects the classification top-1 accuracy. Training loss is used for LSTM as the dataset is unlabelled and has no test set. We simulated the training with nine clients using BatchCrypt's quantization scheme including dACIQ clipping. The simulation scripts are also open-sourced for public access. We set the quantization bit width to 8, 16, and 32, respectively, and compare the results against plain training (no encryption) as the baseline. We ran the experiments until convergence, which is achieved when the accuracy or loss does not reach a new record for three consecutive epochs.

Fig. 6 depicts the results. For FMNIST, plain baseline reaches peak accuracy 88.62% at the 40th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 88.67%, 88.37%, and 88.58% at the 122nd, 68th, and 32nd epoch, respectively. For CIFAR, plain baseline reaches peak accuracy 73.97% at the 285th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 71.47%, 74.04%, and 73.91% at the 234th, 279th, and 280th epoch, respectively. Finally, for LSTM, plain baseline reaches bottom loss 0.0357 at the 20th epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 0.1359, 0.0335, and 0.0386 at the 29th, 23rd, and 22nd epoch, respectively. We hence conclude that, with appropriate quantization bit width, BatchCrypt's quantization has negligible negative impact on the trained model quality. Even in



(a) FMNIST worker

(b) FMNIST aggregator

(c) CIFAR worker

(d) CIFAR aggregator

(e) LSTM worker

(f) LSTM aggregator

Figure 7: Breakdown of training iteration time under stock FATE and BatchCrypt, where "idle" measures the idle waiting time of a worker and "agg." measures the gradient aggregation time on the aggregator. Note that model computation is left out here as it contributes little to the iteration time.

the case where the quantized version requires more epochs to converge, we later show that such overhead can be more than compensated by the speedup from BatchCrypt.

Although 8-bit quantization performs poorly for CIFAR and LSTM, it is worth notice that, longer bit width does not necessarily lead to higher model quality. In fact, quantized training sometimes achieves better results. Prior quantization work has observed similar phenomenon [63], where the stochasticity introduced by quantization can work as a regularizer to reduce overfitting, similar to a dropout layer [54]. Just like the dropout rate, quantization bit width acts as a trade-off knob for how much information is retained and how much stochasticity is introduced.

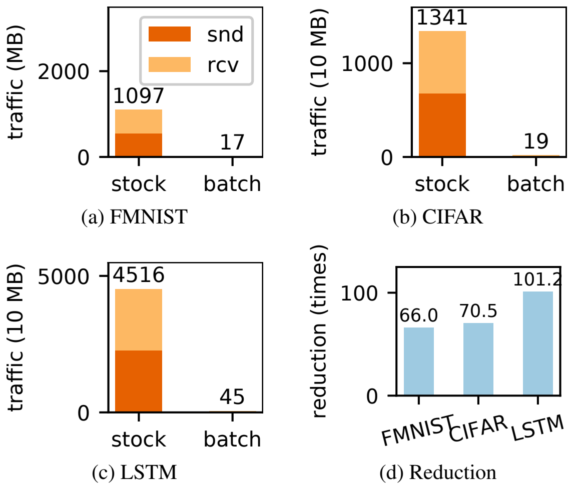
In summary, with apt bit width, our gradient quantization scheme does not adversely affect the trained model quality. In contrast, existing batching scheme introduces 5% of quality drop [37]. Thus, quantization-induced error is not a concern for the adoption of BatchCrypt.

6.3 Effectiveness of BatchCrypt

BatchCrypt vs. FATE We next evaluate the effectiveness of BatchCrypt in real deployment. We set the quantization bit width to 16 as it achieves a good performance (§6.2). The batch size is set to 100, in which we pad two zeros between the two adjacent values. We report two metrics: the iteration time breakdown together with the network traffic. We ran the experiments for 50 iterations, and present the averaged results against those measured with the stock FATE implementation in Figs. 7 and 8. We see in Fig. 2 that BatchCrypt significantly speeds up a training iteration: 23.3× for FMNIST, 70.8× for CIFAR, and 92.8× for LSTM. Iteration time breakdown further shows that our implementation reduces the cost of HE related operations by close to 100×, while the communication time is substantially reduced as well ("idle" in worker and "transfer" in aggregator).

We next refer to Fig. 8, where we see that BatchCrypt reduces the network footprint by up to 66×, 71×, and 101× for FMNIST, CIFAR, and LSTM, respectively. Note that FATE adopts `grpc` as the communication vehicle whose limit on payload forces segmenting encrypted weights into small chunks before transmission. By reducing the size of data to transfer, BatchCrypt alleviates the segmenting induced overhead (metadata, checksum, etc.), so it is possible to observe a reduction greater than the batch size.

Our experiments also show that BatchCrypt achieves more salient improvements for larger models. First, encryption related operations take up more time in larger models, leaving more potential space for BatchCrypt. Second, since layers are batched separately, larger layers have higher chances forming long batches. BatchCrypt's speedup can be up to two orders of magnitude, which easily offset the extra epochs needed for convergence caused by quantization (§6.2).



(a) FMNIST

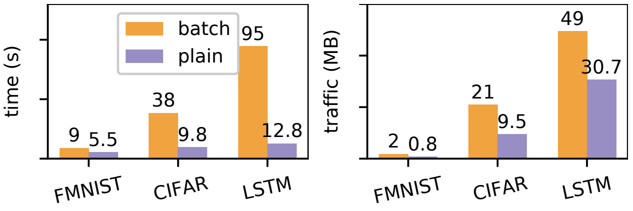
(b) CIFAR

(c) LSTM

(d) Reduction

Figure 8: Comparison of the network traffic incurred in one training iteration using the stock FATE implementation and BatchCrypt.

BatchCrypt vs. Plaintext Learning We next compare BatchCrypt with the plain distributed learning where no encryption is involved—an ideal baseline that offers the optimal performance. Fig. 9 depicts the iteration time and the network



(a) Time

(b) Communication

Figure 9: Time and communication comparisons of one iteration on workers between BatchCrypt and plain distributed learning without encryption.

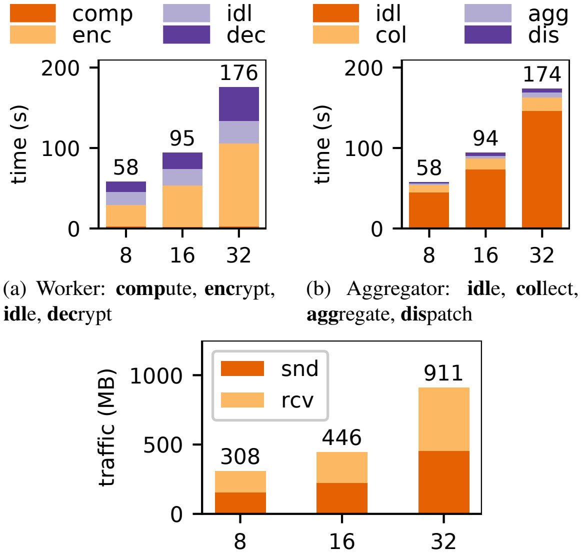
Table 4: Projected total training time and network traffic usage until convergence for the three models. The converged test accuracy for FMNIST, CIFAR as well as loss for LSTM and their corresponding epoch numbers are listed in the table.

Model	**Mode**	**Epochs**	**Acc./Loss**	**Time** (h)	**Traffic** (GB)
FMNIST	stock	40	88.62%	122.5	2228.3
	batch	68	88.37%	8.9	58.7
	plain	40	88.62%	3.2	11.17
CIFAR	stock	285	73.97%	9495.6	16422.0
	batch	279	74.04%	131.3	227.8
	plain	285	73.97%	34.2	11.39
LSTM	stock	20	0.0357	8484.4	15347.3
	batch	23	0.0335	105.2	175.9
	plain	20	0.0357	12.3	10.4

footprint under the two implementations. While encryption remains the major bottleneck, BatchCrypt successfully reduces the overhead by an order of magnitude, making it practical to achieve the same training results as the plain distributed setting. Note that encrypted numbers in FATE each carries redundant information such as public keys, thus causing the communication inflation compared with the plain version. Such inflation can be reduced if FATE employs some optimized implementation.

Training to Convergence Our previous studies mainly focus on a single iteration. Compared with stock FATE and plain distributed learning, BatchCrypt requires a different number of iterations to converge. We hence evaluate their end-to-end performance by training ML models till convergence. As this would take exceedingly long time and high cost if performed in real deployment, we instead utilize our simulation in §6.2 and iteration profiling results to project the total time and network traffic needed for convergence.

Table 4 lists our projection results of the three solutions. Compared with the stock implementation in FATE, BatchCrypt dramatically reduces the training time towards convergence by 13.76×, 72.32×, and 80.65× for FMNIST, CIFAR, and LSTM, respectively. In the meantime, the network footprints shrink by 37.96×, 72.01×, 87.23×, respectively. We stress that these performance improvements are achieved without degrading the trained model quality. On the other hand, BatchCrypt only slows down the overall training time by 1.78×, 2.84×, and 7.55× for the three models compared with plain learning—which requires no encryption and hence achieves the fastest possible training convergence. In summary, BatchCrypt significantly reduces both the computation and communication overhead caused by HE, enabling efficient HE for cross-silo FL in production environments.



(a) Worker: **comp**ute, **enc**rypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

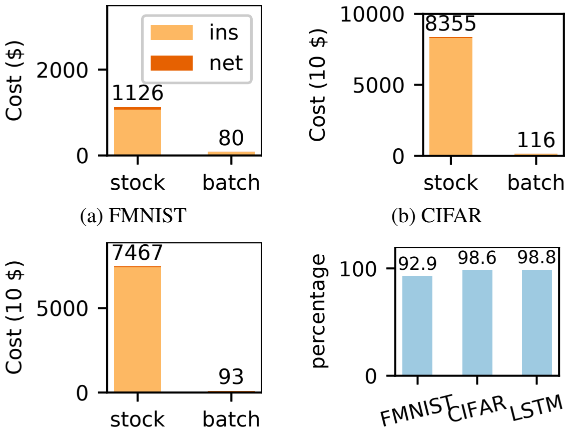
(c) Aggregator traffic

Figure 10: Breakdown of iteration time and communication traffic of BatchCrypt with LSTM model with various quantization bit widths in one iteration. The corresponding batch sizes for bit width 8, 16, and 32 are 200, 100, and 50, respectively.

6.4 Batching Efficiency

We have shown in §6.2 that ML applications have different levels of sensitivity towards gradient quantization. It is hence essential that BatchCrypt can efficiently batch quantized values irrespective of the chosen quantization bit width. Given an HE key, the longest plaintext it can encrypt is determined by the key size, so the shorter the quantization width is, the larger the batch size is, and the higher the potential speedup could be. We therefore look into how our BatchCrypt implementation can exploit such batching speedup.

We evaluate BatchCrypt by varying the batch size. In particular, we train the LSTM model on the geo-distributed clients with different quantization widths 8, 16, and 32. The corresponding batch sizes are set respectively to 200, 100, and 50. We ran the experiments for 50 iterations, and illustrate the average statistics in Fig. 10. Figs. 10a and 10b show the time breakdown in the three experiments. It is clear that employing a shorter quantization bit width enables a larger batch size, thus leading to a shorter training time. Note that the speedup going from 8-bit to 16-bit is smaller compared with that from 16-bit to 32-bit, because HE operations become less of a bottleneck with larger batch size. Fig. 10c depicts the



(a) FMNIST (b) CIFAR (c) LSTM (d) Savings

Figure 11: Total cost until convergence between FATE's stock implementation and BatchCrypt, **ins**tance and **net**work costs are highlighted separately.

accumulated network traffic incurred in one iteration, which follows a similar trend as that of the iteration time. In conclusion, BatchCrypt can efficiently exploit batching thanks to its optimized quantization. Similar to [26, 51], BatchCrypt's batching scheme reduces both the computation and communication cost linearly as the batch size increases. In fact, if lattice-based HE algorithms are adopted, one can replace BatchCrypt's batching scheme with that of [26, 51], and still benefit from BatchCrypt's accuracy-preserving quantization.

6.5 Cost Benefits

The reduced computation and communication overheads enable significant cost savings: sustained high CPU usage leads to high power consumption, while ISPs charge for bulk data transfer over the Internet. As our evaluations were conducted in EC2, which provides a runtime environment similar to the organization's own datacenters, we perform cost analysis under the AWS pricing scheme. The hourly rate of our cluster is \$8.758, while the network is charged based on outbound traffic for \$0.042, \$0.050, \$0.042, \$0.048, \$0.055 per GB for the regions listed in Table 2.

We calculate the total cost for training until convergence in Table 4 and depict the results in Fig. 11. As both computation and communication are reduced substantially, BatchCrypt achieves huge cost savings over FATE. While the instance cost reduction is the same as the overall speedup in Table 4, BatchCrypt lowers the network cost by 97.4%, 98.6% and 98.8% for FMNIST, CIFAR, and LSTM, respectively.

7 Discussion

Local-update SGD & Model Averaging Local-update SGD & model averaging is another common approach to reducing the communication overhead for FL [22, 40], where the aggregator collects and averages model weights before propagating them back to clients. Since there are only addition operations involved, BatchCrypt can be easily adopted.

Split Model Inference In many FL scenarios with restrictive privacy requirement, a trained model is split across clients, and model inference involves coordination of all those clients [23, 61]. BatchCrypt can be used to accelerate the encryption and transmission of the intermediate inference results.

Flexible Synchronization There have been many efforts in amortizing the communication overhead in distributed SGD by removing the synchronization barriers [24, 34, 62]. Although we only evaluate BatchCrypt's performance in synchronous SGD, our design allows it to take advantage of the flexible synchronization schemes proposed in the literature. This is not possible with Secure Aggregation [9].

Potential on Large Models Recent research and our evaluations show that more sophisticated ML models are more resilient to quantization noise. In fact, certain models are able to converge even with 1- or 2-bit quantization [8, 58]. The phenomenon promises remarkable improvement with BatchCrypt, which we will explore in our future work.

Applicability in Vertical FL Vertical FL requires complicated operations like multiplying ciphertext matrices [38, 61]. Batching over such computation is beyond BatchCrypt's current capability. We will leave it as a future work.

8 Concluding Remark

In this paper, we have systematically studied utilizing HE to implement secure cross-silo FL. We have shown that HE related operations create severe bottlenecks on computation and communication. To address this problem, we have presented BatchCrypt, a system solution that judiciously quantizes gradients, encodes a batch of them into long integers, and performs batch encryption to dramatically reduce the encryption overhead and the total volume of ciphertext. We have implemented BatchCrypt in FATE and evaluated its performance with popular machine learning models across geo-distributed datacenters. Compared with the stock FATE, BatchCrypt accelerates the training convergence by up to 81× and reduces the overall traffic by 101×, saving up to 99% cost when deployed in cloud environments.

Acknowledgement

We thank our shepherd, Brandon Lucia, and the anonymous reviewers for their valuable feedbacks that help improve the quality of this work. This work was supported in part by RGC ECS grant 26213818, WeBank-HKUST research collaboration grant 2019, NSF CCF-1756013 and NSF IIS-1838024. Chengliang Zhang was supported by the Hong Kong PhD Fellowship Scheme.

References

[1] ALISTARH, D., GRUBIC, D., LI, J., TOMIOKA, R., AND VOJNOVIC, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NeurIPS* (2017).

[2] ANDERSON, A. G., AND BERG, C. P. The high-dimensional geometry of binary neural networks. In *ICLR* (2018).

[3] Aws deep learning ami. https://aws.amazon.com/machine-learning/amis/, 2019.

[4] BANNER, R., HUBARA, I., HOFFER, E., AND SOUDRY, D. Scalable methods for 8-bit training of neural networks. In *NeurIPS* (2018).

[5] BANNER, R., NAHSHAN, Y., AND SOUDRY, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *NeurIPS* (2019).

[6] BARKER, E., BARKER, W., BURR, W., POLK, W., AND SMID, M. Recommendation for key management part 1: General (revision 3). *NIST special publication 800*, 57 (2012), 1–147.

[7] BASKIN, C., SCHWARTZ, E., ZHELTONOZHSKII, E., LISS, N., GIRYES, R., BRONSTEIN, A. M., AND MENDELSON, A. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *arXiv preprint arXiv:1804.10969* (2018).

[8] BERNSTEIN, J., WANG, Y.-X., AZIZZADENESHELI, K., AND ANANDKUMAR, A. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434* (2018).

[9] BONAWITZ, K., IVANOV, V., KREUTER, B., MARCEDONE, A., MCMAHAN, H. B., PATEL, S., RAMAGE, D., SEGAL, A., AND SETH, K. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1175–1191.

[10] BOTTOU, L., AND BOUSQUET, O. The tradeoffs of large scale learning. In *NeurIPS* (2008).

[11] BRAKERSKI, Z., GENTRY, C., AND VAIKUNTANATHAN, V. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT) 6*, 3 (2014), 1–36.

[12] California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa, 2018.

[13] CHENG, K., FAN, T., JIN, Y., LIU, Y., CHEN, T., AND YANG, Q. Secureboost: A lossless federated learning framework. *arXiv preprint arXiv:1901.08755* (2019).

[14] COURBARIAUX, M., BENGIO, Y., AND DAVID, J.-P. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[15] DATA61, C. Python paillier library. https://github.com/data61/python-paillier, 2013.

[16] DU, W., HAN, Y. S., AND CHEN, S. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *Proceedings of the 2004 SIAM international conference on data mining* (2004), SIAM, pp. 222–233.

[17] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.

[18] FATE (Federated AI Technology Enabler). https://github.com/FederatedAI/FATE, 2019.

[19] GE, T., AND ZDONIK, S. Answering aggregation queries in a secure system model. In *VLDB* (2007).

[20] Tensorflow Federated. https://www.tensorflow.org/federated, 2019.

[21] GUPTA, S., AGRAWAL, A., GOPALAKRISHNAN, K., AND NARAYANAN, P. Deep learning with limited numerical precision. In *ICML* (2015).

[22] HADDADPOUR, F., KAMANI, M. M., MAHDAVI, M., AND CADAMBE, V. Local sgd with periodic averaging: Tighter analysis and adaptive synchronization. In *NeurIPS* (2019).

[23] HARDY, S., HENECKA, W., IVEY-LAW, H., NOCK, R., PATRINI, G., SMITH, G., AND THORNE, B. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677* (2017).

[24] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *NeurIPS* (2013).

[25] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[26] JUVEKAR, C., VAIKUNTANATHAN, V., AND CHANDRAKASAN, A. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1651–1669.

[27] KAIROUZ, P., MCMAHAN, H. B., AVENT, B., BELLET, A., BENNIS, M., BHAGOJI, A. N., BONAWITZ, K., CHARLES, Z., CORMODE, G., CUMMINGS, R., ET AL. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977* (2019).

[28] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[29] KOLOSKOVA, A., STICH, S. U., AND JAGGI, M. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *ICML* (2019).

[30] KONEČNÝ, J., MCMAHAN, H. B., RAMAGE, D., AND RICHTÁRIK, P. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527* (2016).

[31] KRIZHEVSKY, A., HINTON, G., ET AL. Learning multiple layers of features from tiny images. Tech. rep., Citeseer, 2009.

[32] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *NeurIPS* (2012).

[33] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014), USENIX.

[34] LIAN, X., HUANG, Y., LI, Y., AND LIU, J. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NeurIPS* (2015).

[35] LIN, T., STICH, S. U., PATEL, K. K., AND JAGGI, M. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217* (2018).

[36] LIN, Y., HAN, S., MAO, H., WANG, Y., AND DALLY, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[37] LIU, C., CHAKRABORTY, S., AND VERMA, D. Secure model fusion for distributed learning using partial homomorphic encryption. In *Policy-Based Autonomic Data Governance*. Springer, 2019, pp. 154–179.

[38] LIU, Y., CHEN, T., AND YANG, Q. Secure federated transfer learning. *arXiv preprint arXiv:1812.03337* (2018).

[39] MCMAHAN, H. B., MOORE, E., RAMAGE, D., HAMPSON, S., ET AL. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).

[40] MCMAHAN, H. B., MOORE, E., RAMAGE, D., AND Y ARCAS, B. A. Federated learning of deep networks using model averaging. *ArXiv abs/1602.05629* (2016).

[41] MIGACZ, S. 8-bit inference with tensorrt. In *GPU technology conference* (2017), vol. 2, p. 7.

[42] Mohassel, P., and Rindal, P. Aby 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 35–52.

[43] Mohassel, P., and Zhang, Y. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 19–38.

[44] Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., and Taft, N. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 334–348.

[45] Cybersecurity Law of the People's Republic of China. http://www.lawinfochina.com/display.aspx?id=22826&lib=law, 2017.

[46] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), Springer, pp. 223–238.

[47] Pathak, M., Rane, S., and Raj, B. Multiparty differential privacy via aggregation of locally trained classifiers. In *NeurIPS* (2010).

[48] Phong, L. T., Aono, Y., Hayashi, T., Wang, L., and Moriai, S. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security 13*, 5 (2018), 1333–1345.

[49] Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., and Passerat-Palmbach, J. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017* (2018).

[50] San, I., At, N., Yakut, I., and Polat, H. Efficient paillier cryptoprocessor for privacy-preserving data mining. *Security and communication networks 9*, 11 (2016), 1535–1546.

[51] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, Apr. 2020. Microsoft Research, Redmond, WA.

[52] Shokri, R., and Shmatikov, V. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), ACM, pp. 1310–1321.

[53] Soudry, D., Hubara, I., and Meir, R. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NeurIPS* (2014).

[54] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[55] Text generation with an rnn. https://www.tensorflow.org/tutorials/text/text_generation, 2019.

[56] Wang, J., and Joshi, G. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313* (2018).

[57] WeBank. https://www.webank.com/en/, 2019.

[58] Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *NeurIPS* (2017).

[59] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *NeurIPS* (2017), pp. 4148–4158.

[60] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[61] Yang, Q., Liu, Y., Chen, T., and Tong, Y. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST) 10*, 2 (2019), 12.

[62] Zhang, C., Tian, H., Wang, W., and Yan, F. Stay fresh: Speculative synchronization for fast distributed machine learning. In *ICDCS* (2018), IEEE.

[63] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

A Deep Dive into DNS Query Failures

Donghui Yang[†] [§], Zhenyu Li[†] [§] [‡], Gareth Tyson[♭]
[†]*ICT-CAS,* [§]*University of Chinese Academy of Sciences,* [‡] *Purple Mountain Laboratories,* [♭]*QMUL*

Abstract

The Domain Name System (DNS) is fundamental to the operation of the Internet. Failures within DNS can have a dramatic impact on the wider Internet, most notably preventing access to any services dependent on domain names (*e.g.* web, mobile apps). Although there have been several studies into DNS utilization, we argue that greater focus should be placed on understanding *how* and *why* DNS queries fail in-the-wild. In this paper, we perform the largest ever study into DNS activity, covering 3B queries. We find that 13.5% of DNS queries fail, and this leads us to explore the root causes. We observe significant differences between IPv4 and IPv6 lookups. A handful of domains that have high failure rates attract a huge volume of queries, and thus dominate the failures. This is particularly the case for domains that are classified as malicious. The success rates also vary greatly across resolvers due to the differences in the domains that they serve and the infrastructure reliability.

1 Introduction

The Domain Name System (DNS) is organized as a distributed system that provides mappings between human-readable domain names (*e.g.* foo.com) and their associated DNS records [17–20]. These include A type records for IPv4 addresses, AAAA for IPv6 addresses, MX for SMTP mail exchanges, NS for name servers, PTR for pointers of reverse DNS lookups and CNAME for domain name aliases. Nearly all Internet-connected applications depend on DNS. As such, it is a critical dependency, whose failure has the potential to create global Internet outages. For example, in 2016, Dyn (a DNS operator) suffered a major Denial of Service attack against its infrastructure. This meant that DNS queries for popular domains such as Netflix and Visa began to fail, crippling access to these services (even though those services were still online).

Although there is a significant body of research into DNS behavior, we argue that *DNS failures* specifically require further investigation. Root DNS server behavior was examined in [5, 8], where negative DNS answers were analyzed including NXDOMAIN responses. Callahan *et al.* [4] also examined the DNS behavior from the perspectives of performance and response message. We differ in that our focus is on failures not caused by NXDOMAINs. In addition, we note that DNS has evolved significantly since these studies, *e.g.* the rise of new gTLDs and IDNs. Although there have been a number of studies of new gTLDs and IDNs [9, 13, 15], they mainly focus on domain registration behavior and cyber attacks, while we complement these studies with DNS query failure analysis. Other work [12, 22, 25] has leveraged NXDOMAIN responses to detect botnets or DGAs. Again, our work focuses on failures caused by DNS infrastructures instead of NXDOMAINs.

With the above in-mind, we present a large-scale analysis of DNS query failures in-the-wild. To achieve this, we gather a unique dataset containing 3B DNS queries (Section 2). We find that failed queries are, indeed, common place with 13.5% of all queries not successfully resolved. This motivates us to inspect which factors correlate most closely with failed queries (Section 3). We observe a highly skewed distribution, whereby a small number of domains are responsible for the majority of failures. AAAA queries (IPv6) are particularly unreliable, with only 1/3 of queries successfully resolved. This is perhaps understandable given the use of protocols such as Happy Eyeballs [24], although we also find that many domains lack AAAA support. We further inspect the relationship between failures and the DNS resolver used, to find a vast array of resolvers, with 13.5% of queries in our dataset being issued to public resolvers, *e.g.* OpenDNS. We observe diverse failure rates across the resolvers, confirming that they do have an impact on failures. We also note differences among the Top Level Domains (TLDs) with, for example, the new wave of generic-TLDs having higher failure rates than more traditional TLDs. Finally, we propose system implications based on our findings. To sum up, we make the following key findings:

- In spite of the promotion of IPv6 over recent years [7, 21, 26], the majority (86.2%) of DNS queries are still for A records, while only 10.4% are for AAAA records (comparable to the proportion in 2012 [8]). The failure rate for A lookups is 6.9%, yet, to our surprise, the failure rate for AAAA queries is as high as 64.2%, almost 3 times of that in 2012 [8]. We observe that approximately 60% of domains do not support AAAA queries.

- We explore a number of factors to explore the causes of failures. We observe a heavy-tailed distribution of failures across domains, indicating that a handful of domains contribute to most of the failures. 20% of local resolvers in our dataset have never successfully resolved AAAA queries, implying they are not ready for IPv6. The use of public resolvers may also impact query failures as they show diverse failure rates, in-part due to the distinct domain sets that each serves and the differences in infrastructure reliability.
- The success rates for new gTLD domains and IDNs are 10% lower than that of well established domains. This is largely because of the prevalence of malicious domains. Certain ASes are prevalent for hosting malicious new gTLD domains, and these new gTLD domains contribute to 73.7% of the all new gTLD queries. The malicious domains are, however, volatile and change frequently. In fact, *none* are resolvable today. The malicious new gTLD domains in these ASes are of various types and have distinct network footprints.

2 Dataset

Dataset Oveview. Our dataset consists of passive DNS logs that are generated by Deep Packet Inspection (DPI) appliances in 3 ISPs in China. Each DPI appliance parses the DNS response messages from recursive resolvers to end users, and generates a log for each response. A log includes the end user's anonymized IP address, BGP prefix,[1] the ASN (Autonomous System Number), the recursive resolver's IP address, the DNS query type, all the resource records, the timestamp (in seconds) and an indicator about whether the resolver and the end user's original IP address share a common /24 prefix. In cases of CNAME responses, we follow the redirection to the final record. The dataset contains 14 samples that were collected every other day in February 2018. Each sample consists of 10-minute logs generated by all the DPI appliances of the 3 ISPs. In total, we obtain 3,085,998,589 logs. It is noteworthy that while there were IPv6 addresses in the response IP list of AAAA queries, no IPv6 addresses were seen in end users' IP addresses and recursive DNS resolvers.

Identification of Failed Queries. We next extract the set of failed queries for the four most popular types of records (A, AAAA, PTR, MX), because they constitute 99.5% of all queries. For each response, we extract the requested domain (the QNAME) from the Question portion, and check if the response contains a valid answer (*e.g.* for an A query, at least one RR in the response is an A record of the requested domain). In this paper, we are interested in failures caused by DNS infrastructures instead of NXDOMAINs (*e.g.* typos). However, we do not have the response code (*e.g.* 'NOERROR', 'NXDOMAIN' or other status) in our dataset. Therefore, we turn to a heuristic method to filter out logs that are attributed to NXDOMAINs. Specifically, for each domain requested (*i.e.,*QNAME in the log), we check if it has succeeded at least once in our logs. We then remove the logs containing domains that have never succeeded in the whole dataset, as they are likely NXDOMAINs. The subsequent analyses are based on the remaining dataset, which contains 2,811,010,890 logs issued by 37,070,965 unique IP addresses to 246,991 resolvers.

Caveats. It is important to highlight potential limitations in our data. The above heuristic method may leave some domains that were resolvable at a time but then became NXDOMAINs later. Moreover, our dataset does not allow us to inspect failed queries that did not trigger a response (*e.g.* due to packet loss). Naturally, the fact that a DNS response is returned does not necessarily mean that the web server is live and responsive. Therefore, we only inspect if a valid DNS response is returned (not if the IP address is correct). There are many possibilities that lead to incorrect mapping of domains to addresses, such as DNS manipulation [16] and on-path DNS interception [14]. Another related concern pertains to censored domains. Thus, before continuing, we test if a censored domain will return a valid IPv4/IPv6 address for an A/AAAA query. Our tests confirm that, indeed, valid addresses are returned, even when querying censored websites.[2] Another potential limitation is that our data is local to China. Nevertheless, we believe the scale of the Chinese Internet means that these findings can still have a major impact. As DNS is a globally distributed system where China and other countries are all involved, there is not much specific to China from the perspective of the DNS infrastructure. We also note that (to the best of our knowledge) this is by far the largest DNS failure dataset ever studied.

Ethical Issues. The ISPs collect the DNS logs for the purpose of improving their service quality and security. The end users' IP addresses were anonymized and we are unable nor allowed to link queries to users. Users are notified when subscribing that the ISPs may collect this information, and may share it with academics for research. Our study has not triggered the collection of any new data. All data was processed in a secure silo by the first author.

3 Exploring DNS Query Failures

3.1 A Primer on DNS Failures

We begin by simply computing the number and types of failed queries in our dataset. Table 1 shows the percentage of query

[1] IP addresses and BGP prefixes are anonymized with `Crypto-PAn` [2]

[2] Since we focus on DNS query failures, we did not check whether the returned IP address does host the queried domain or not [1].

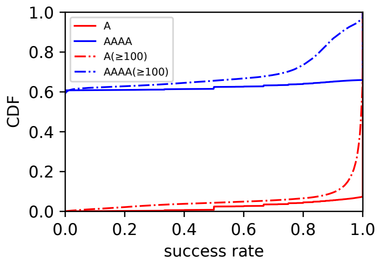


Figure 1: CDFs of success rates of domains for A and AAAA queries.

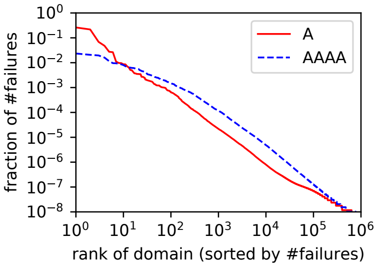


Figure 2: #failures - rank of domains(log-log).

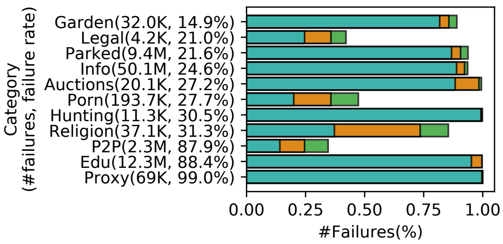


Figure 3: The top 11 domain categories sorted by failure rates with # of failures $>$ 1K.

types, alongside the *overall* success rate, which is 1 minus the ratio of failed queries to all queries of each query type. We present the four most popular query types. The A type queries account for the majority (86.2%), and AAAA queries have a smaller query volume (10.4%).[3] We observe a variety of success rates across the query types. Overall, the A queries are successfully resolved most frequently, while other query types manifest lower success rates. This confirms that a sizeable fraction of queries are *not* successfully resolved; we spend the rest of the section exploring factors influencing this trend.

Table 1: Four popular types of DNS queries: percentages of the number of queries and success rates.

Query Type	A	AAAA	PTR	MX	Others
#queries	86.2%	10.4%	2.8%	0.1%	0.5%
Success Rate	93.1%	35.8%	40.4%	82.9%	-

3.2 Failures Across Domains

We first compute the distribution of failures across domains. Due to their prominence, we focus on A and AAAA queries. Figure 1 presents the CDFs of the success rates across domains encountered within our dataset.

A Queries. A queries exhibit high success rates: overall, 93.7% of domains have a success rate exceeding 95% suggesting high reliability. There are ouliers though; for instance, the bottom 0.1% of domains have success rates below 5%. To eliminate the impact of low-frequency domains on the results, we filter out domains that issue fewer than 100 requests and plot the CDF of success rate of the remaining domains (the red dash-dot line), where 84.9% of remaining domains have a success rate exceeding 95%. Nevertheless, as many as 7% of domains experience a success rate lower than 50%. Given that we have removed the failures caused by NXDOMAINs, the result suggests that problems with the DNS infrastructure do impact users when visiting these domains.

AAAA Queries. For AAAA queries, only 34.3% of domains have a success rate exceeding 95%. When limiting to domains whose query frequency exceeds 100, only 7.8% of domains have a success rate exceeding 95%, while about 60% of domains have never been successfully resolved. Again, given that we only include domains that have been successfully resolved (considering all query types), this suggests that there are infrastructural limitations in how DNS supports IPv6.

Domain Failure Rates The above suggests that the majority of failures are the responsibility of a small set of domains, especially for A queries. To explore this further, Figure 2 presents the number of failures per domain on a log-log plot. We sort the X-axis by the rank of the domain (based on the number of failures). We see that failures are concentrated on a small number of domains. To gain a further understanding of the types of domains that have high failure rates, we utilize the Webroot Brightcloud API[4] to classify the top 50K domains (measured by failure rate). Figure 3 presents the results. The Y-axis shows the top 11 categories sorted by failure rate, where the number of failed queries ($>$ 1K) and the failure rate is shown in the parentheses. For each category, we plot the ratio of the number of failed queries of the top 3 SLDs to the total number of failures. A number of classifications which can be expected to fail frequently are present, *e.g.* proxy, porn and parked domains. This suggests that such domain types are paramount in increasing failure rates. However, it is unexpected to see the Education category is ranked second. Closer inspection reveals that *clock.cuhk.edu.hk*, which is the third most failed domain, contributes the most failures. Another interesting observation is the concentration of failures for each category on a few domains. For 8 out of the 11 categories, over 80% of the failures are attributed to the top 3 SLDs (top 1 SLD in most cases).

3.3 Failures Across Resolvers

Another explanation for failed queries is that the resolvers may not correctly handle queries.

Testing Resolvers. To explore this, we calculate the success rate of queries issued to each DNS resolver (identified by the resolver's IP address). Figure 4 presents the CDF of the success rate for the domains per resolver.[5] The majority of

[3] Note that we follow CNAME redirections, rather than reporting them here as responses.

[4] https://www.brightcloud.com/web-service

[5] We eliminate the resolvers serving fewer than 100 queries in our dataset to avoid bias.

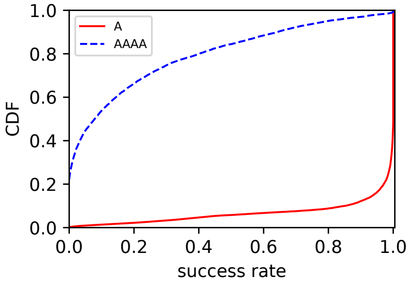


Figure 4: CDF of the success rate for individual resolvers.

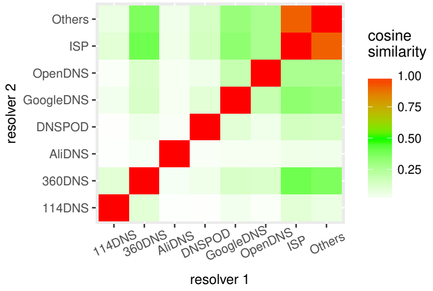


Figure 5: Cosine similarity between each pair of DNS resolver types.

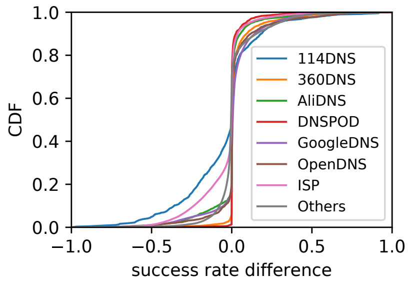


Figure 6: Different success rates of same domains handled by different resolvers.

Table 2: The number of A and AAAA queries and their success rates (shown in the parentheses) handled by each resolver.

	114DNS	360DNS	AlibabaDNS	DNSPOD	GoogleDNS	OpenDNS	ISP	Others
A	296.4K(98.5%)	831.0K(95.9%)	667.8K(94.7%)	352.5K(99.6%)	333.4M(90.7%)	467.6K(86.3%)	48.7M(95.3%)	2.1B(93.5%)
AAAA	75.4K(14.5%)	50.3K(61.8%)	112.9K(52.4%)	15.5K(54.3%)	40.6M(43.4%)	31.0K(49.2%)	9.6M(22.8%)	252.6M(35.0%)

resolvers have very high success rates when serving A queries: about half experience almost no failures. By contrast, 60% of resolvers serving AAAA queries can successfully resolve just 20% of the queries. Surprisingly, about 20% of the resolvers never succeed in resolving AAAA queries. These resolvers may not be IPv6 ready during our observation period.

Testing Public Resolvers. Closer inspection reveals that a notable set of queries are sent to *public resolvers* [3,6,10]. Hence, we also inspect the reliability of these public infrastructures, which include 114 (Chinese) DNS resolvers provided by multiple telecom operators, DNS Pai which belongs to Qihoo 360, AliDNS which belongs to Alibaba, and DNSPOD which belongs to Tencent. We also take into account GoogleDNS and OpenDNS which are widely used throughout the world. We identify these public DNS resolvers by the IP addresses offered on their official websites. Table 2 shows the number of A and AAAA queries handled by each public DNS resolver mentioned above along with their success rates. GoogleDNS dominates the most used public DNS service (even though Google is less well known in China). Others do not show much difference in terms of query volume. We observe various success rates across public DNS resolvers though. For example, DNSPOD succeeds in almost all its A queries, while OpenDNS achieves just 86.3%. There is also notably lower success rate across all resolvers for AAAA queries.

The above confirms that resolvers do seem to have an impact on success rates. A potential reason for this is that the resolvers may simply receive different queries. To explore this, we compute a vector for each resolver, where each element represents a domain, which appears in A or AAAA queries using the resolver, and the query volume of that domain handled by the resolver. Then we calculate the similarity of each pair of DNS resolver using the cosine similarity between their vectors. Figure 5 illustrates the result. The resolvers actually demonstrate a surprisingly low similarity with each other, signalling rather different request patterns. Among these resolvers, 114DNS and AliDNS are the least like the others. Indeed, 114DNS handles many requests for Akamai domains which appear less often in other resolvers, while AliDNS handles many requests of taobao.com and alipay.com which belongs to Alibaba services. This could be the reason for the variance of success rates observed from different resolvers.

Another possible explanation is the differences between resolvers' infrastructures. To explore this, we compare the success rates of the same domains handled by different resolvers. Specifically, for each resolver, we first find the domain intersection of it and each other resolver. Then for each domain in each intersection, we calculate the difference in their success rates on the two resolvers. Finally, for each type of resolver, we plot the CDF of the differences between this type to other types in Figure 6. Note, a difference below 0 indicates a lower success rate of this type of resolver, and a positive value indicates a higher success rate. We can see significant differences for some types of resolvers: domains resolved by 114DNS and ISP are more likely to fail, while DNSPOD and 360DNS have higher success rates. This observation partially explains the results in Table 2.

3.4 Failures Across TLDs

We next inspect if certain TLDs have lower query success rates. Specifically, we explore two camps of TLDs: the new generic Top Level Domains (gTLD),[6] and those that have Internationalized Domain Name (IDNs). Our dataset contains 611,769 new gTLDs and 79,705 IDNs. Table 3 summarizes our results. We see rather different rates of success across the domain and query types. The lower success rate for new gTLDs may be because such gTLDs attract certain types of domain registrant. For example, the .lol domain is well known to attract large volumes of malicious activites [11]. With this in-mind, we find a success rate of just 20.3% for

[6] Based on the list from nTLDStats https://ntldstats.com

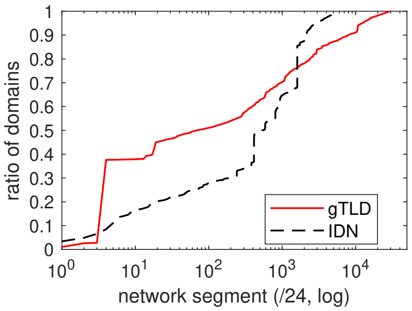


Figure 7: Distri. of new gTLD domains (gTLD in the legend for short) and IDNs seperated by /24 network segments.

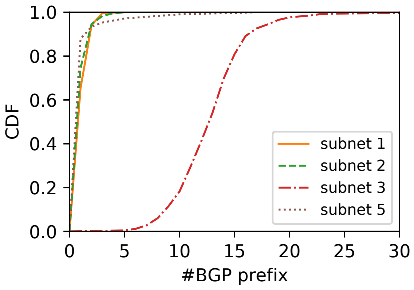


Figure 8: CDF of # BGP prefixes requesting individual malicious SLDs.

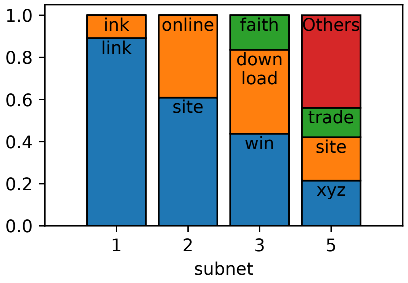


Figure 9: TLDs and their fractions of # malicious SLDs.

.lol, compared to 83.0% of .com. Another reason for failed queries is the presence of malicious domains which are unreliable. To inspect this, we extract all A record queries that were successfully resolved and investigate the network segments hosting them. Overall, we obtain 113,539 (11,729) IP addresses hosting new gTLDs (IDNs) mapping to 29,047 (5,635) /24 network segments, respectively.

Table 3: The number of queries and success rates (shown in the parentheses) of new gTLD domains and IDNs.

	new gTLD	IDN
total	4.0M (79.3%)	0.26M (66.6%)
A	3.4M (88.6%)	0.17M (86.7%)
AAAA	0.6M (25.9%)	0.09M (26.4%)

Figure 7 presents the distribution of the number of new gTLD domains and IDNs per network segment (sorted by #IPs hosting new gTLD domains and IDNs, respectively). Several surges in both lines further arouse our attention: they indicate the existence of some /24 network segments that serve a large number of new gTLD domains or IDNs. This is naturally driven by the presence of large web hosting providers.

Thus, we extract the top 5 surges for both new gTLD domains and IDNs to explore their details. Due to space limitation, we only present the results of new gTLD domains in Table 4, where the last column presents the number of domains that are resolvable on 26 Sept. 2019. Across all of these top ASes we witness an extremely low rate of successful resolutions. In the most extreme case, we observe 201K queries for 195K domains being mapped to Enzu, none of which are resolvable today. In addition, the number of queries is close to the number of FQDNs, suggesting that these domains are short-lived and change frequently. The above trends lead us to hypothesize that some of these domains may be associated with malicious activities. To explore this, we leverage two blacklists from VirusTotal and Qihoo 360 to check the domains. We label a domain as malicious if any of these two blacklists classify it as so. Due to the large volume of domains, we only check SLDs (as opposed to FQDNs). The results are listed in parentheses in Table 4. None of the IDNs are classified as malicious by the two blacklists, however, a *significant* fraction of the new gTLD domains fall into this category. For example, 80% of the SLDs hosted in 23.245.136.0/24 prefix (first row) are classified as malicious. This is common across all of the top new gTLD domains. In total, 73.7% of the queries are for the malicious domains.

Figure 8 presents the distribution of the number of end users' BGP prefixes requesting each malicious SLD in Subnet 1, 2, 3, 5 (shown in Table 4). Except the SLDs hosted in subnet 3, other malicious SLDs affect only 1 or 2 networks. In contrast, malicious SLDs hosted in subnet 3 have a footprint in dozens of networks, implying a larger impact. One possible explanation is that the subnets host different sites. Hence, Figure 9 presents the make-up of the 5 subnets, confirming that they do map to different TLDs.

4 Implications on Systems Design

In this section, we discuss about the implications of our findings on system design, *i.e.,*what systems we could build based on our observations.

Active Measurement System. Our results show that although IPv6 has been promoted in recent years, AAAA queries still fail frequently, and there exist resolvers that do not support AAAA queries. In order to understand which resolvers support AAAA queries, we can build a system to actively measure the IPv6 support of the resolvers. For instance, similar to [23], we can build a single-node measurement system for monitoring IPv6 support of DNS resolvers. The system can distinguish between resolvers that support and do not support AAAA queries by sending DNS queries of popular domains that support AAAA queries. We can also test whether a domain supports AAAA queries by sending requests of this domain to resolvers that are classified as supporting AAAA queries. Considering the differences between resolvers, we could measure the success rates of domains by sending queries to different resolvers, and use the result to help choose the better resolvers.

In addition, we could compare different resolvers from the perspective of localization, *i.e.,*whether the resolver directs

Table 4: new gTLD domains hosted by the top 5 subnets. The number of domains labeled as malicious are within the parentheses.

No.	subnet	AS num.	AS name	#IPs	#queries	#FQDN	#SLD	#resolvable
1	23.245.136.0/24	18978	Enzu Inc	252	201.9K (157.1K)	195.9K (152.2K)	483 (386)	0(0)
2	192.238.167.0/24	395954	Leaseweb	236	17.4K (14.8K)	16.3K (13.9K)	287 (243)	0 (0)
3	172.246.207.0/24	18978	Enzu Inc	236	15.7K (15.4K)	13.2K (13.0K)	443(434)	1 (1)
4	104.217.93.0/24	40676	Psychz Net	253	9.0K (1)	8.8K (1)	923 (1)	9(0)
5	47.89.58.0/24	45102	Alibaba	4	10.9K (469)	8.8K (114)	7.7K (107)	748 (7)

Table 5: The localization performance of resolvers: the proportion of queries directed to servers in the same AS as the end user when possible.

114DNS	360DNS	AlibabaDNS	DNSPOD	GoogleDNS	OpenDNS	ISP	Others
91.2%	98.0%	95.9%	94.3%	64.6%	43.8%	71.7%	69.9%

users to remote servers. This additional function is motivated by our observation presented in Table 5: We calculate for each resolver the fraction of queries that redirect clients to servers in the same AS (when possible). For each domain i, we count all its response IP addresses in the entire dataset and these IP addresses form a set S_i. Then for each log whose QNAME equals i, if at least one IP address in S_i is in the same AS as the end user, we label this query as "possible to be served locally"; if at least one IP address in the response IP addresses of this log is in the same AS as the end user, we label this query as "served locally". These two labels are independent and a log can have both labels, one of the labels, or no label. We aggregate the logs according to the resolvers, and calculate the ratio of *the number of logs labeled as served locally* to *the number of logs labeled as possible to be served locally* for each resolver. We observe that the obtained ratio differs significantly across the resolvers, which indicates that users can choose appropriate resolvers for better network performance. Therefore, it is useful to develop a system for end users to measure the localization performance of different resolvers.

Such an active measurement system is useful for content publishers, ISPs and end users. Many CDNs are being upgraded for better IPv6 support, however, if AAAA queries frequently fail, then the content publisher should be careful to use such CDN IPv6 service. Therefore, it is useful for publishers to locate their content if they can understand in advance which resolvers do not support AAAA queries. In addition, ISPs could also benefit when considering IPv6 network expansion, because understanding which domains support AAAA queries is useful for estimating the IPv6 traffic. For users, the measurement of different resolvers can help them to choose more suitable resolvers considering both IPv6 support and localization performance.

Malicious New gTLD Domain Detection System. We have found that malicious SLDs (of new gTLD domains) hosted by particular ASes contribute to higher failure rates. Manual inspection further reveals that the length of the SLDs tend to be short. Table 6 presents the fraction of malicious SLDs of different length. In more traditional TLDs, malicious domains are usually long because registering a short domain name would cost too much for an attacker. However, registering short new gTLD domains is much easier. Therefore, extracting features from domain names may not work well for detecting malicious new gTLD domains. We could use features like DNS query frequency, the number of FQDNs of an SLD, the resolved IP addresses and the corresponding ASes to build a system for detecting malicious SLDs of new gTLD domains.

Table 6: Fraction of malicious SLDs of different lengths.

Length	3	4	5	≥6
% of SLDs	0.1%	93.0%	6.1%	0.8%

5 Conclusion

The paper has presented a deep dive into DNS query failures using over 3B queries. We have identified high failure rates: 6.9% of A and 64.2% of AAAA queries. IPv6 is far from ready as over half of the domains and 20% of local resolvers do not support AAAA queries. Upgrading these resolvers and popular domains for IPv6 is the first step towards the wider usage of IPv6. Internet users, on the other hand, should be aware of the impact of using public resolvers, from both the perspectives of query failures and mapping inaccuracy [6]. We also found that the volatility of malicious domains (particularly new gTLD domains and IDNs) contributes to higher failure rates because they change frequently and accesses to them results in failures. The corresponding SLDs of malicious new gTLD domains and IDNs, however, are limited, and they are likely hosted by particular ASes. We finally proposed two potential systems that could build on our findings.

Acknowledgement

This work was supported in part by National Key RD Program of China: 2018YFB1800201, the NSF of China (NSFC): 61725206, the Youth Innovation Promotion Association CAS. The corresponding author is Zhenyu Li.

References

[1] Towards a comprehensive picture of the great firewall's DNS censorship. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, San Diego, CA, 2014. USENIX Association.

[2] Crypto-pan. https://www.cc.gatech.edu/computing/Networking/projects/cryptopan/, 2018.

[3] Bernhard Ager, Wolfgang Mühlbauer, Georgios Smaragdakis, and Steve Uhlig. Comparing DNS resolvers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 15–21. ACM, 2010.

[4] Thomas Callahan, Mark Allman, and Michael Rabinovich. On modern DNS behavior and properties. *ACM SIGCOMM Computer Communication Review*, 43(3):7–15, 2013.

[5] Sebastian Castro, Duane Wessels, Marina Fomenkov, and Kimberly Claffy. A day at the root of the internet. *ACM SIGCOMM Computer Communication Review*, 38(5):41–46, September 2008.

[6] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review*, 45(4):167–181, 2015.

[7] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. Measuring IPv6 adoption. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 87–98. ACM, 2014.

[8] Hongyu Gao, Vinod Yegneswaran, Yan Chen, Phillip Porras, Shalini Ghosh, Jian Jiang, and Haixin Duan. An empirical reexamination of global DNS behavior. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 267–278. ACM, 2013.

[9] Tristan Halvorson, Matthew F Der, Ian Foster, Stefan Savage, Lawrence K Saul, and Geoffrey M Voelker. From. academy to. zone: An analysis of the new TLD land rush. In *Proceedings of the 2015 Internet Measurement Conference*, pages 381–394. ACM, 2015.

[10] Cheng Huang, David A Maltz, Jin Li, and Albert Greenberg. Public DNS system and global traffic management. In *2011 Proceedings IEEE INFOCOM*, pages 2615–2623. IEEE, 2011.

[11] Damilola Ibosiola, Ignacio Castro, Gianluca Stringhini, Steve Uhlig, and Gareth Tyson. Who watches the watchmen: Exploring complaints on the web. *Web Conference*, 2019.

[12] N. Jiang, J. Cao, Y. Jin, L. E. Li, and Z. Zhang. Identifying suspicious activities through DNS failure graph analysis. In *The 18th IEEE International Conference on Network Protocols*, pages 144–153, Oct 2010.

[13] Maciej Korczynski, Maarten Wullink, Samaneh Tajalizadehkhoob, Giovane Moura, Arman Noroozian, Drew Bagley, and Cristian Hesselman. Cybercrime after the sunrise: A statistical analysis of DNS abuse in new gTLDs. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 609–623. ACM, 2018.

[14] Baojun Liu, Chaoyi Lu, Haixin Duan, Ying Liu, Zhou Li, Shuang Hao, and Min Yang. Who is answering my queries: Understanding and characterizing interception of the DNS resolution path. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1113–1128, 2018.

[15] Baojun Liu, Chaoyi Lu, Zhou Li, Ying Liu, Hai-Xin Duan, Shuang Hao, and Zaifeng Zhang. A reexamination of internationalized domain names: The good, the bad and the ugly. 2018.

[16] Paul Pearce, Ben Jones, Frank Li, Roya Ensafi, Nick Feamster, Nick Weaver, and Vern Paxson. Global measurement of DNS manipulation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 307–323, 2017.

[17] P.Mockapetris. Domain names–concepts and facilities, rfc 882. http://www.ietf.org/rfc/rfc882.txt, 1983.

[18] P.Mockapetris. Domain names–implementation and specification, rfc 883. http://www.ietf.org/rfc/rfc883.txt, 1983.

[19] P.Mockapetris. Domain names–concepts and facilities, rfc 1034. http://www.ietf.org/rfc/rfc1034.txt, 1987.

[20] P.Mockapetris. Domain names–implementation and specification, rfc 1035. http://www.ietf.org/rfc/rfc1035.txt, 1987.

[21] Enric Pujol, Philipp Richter, and Anja Feldmann. Understanding the share of IPv6 traffic in a dual-stack ISP. In Mohamed Ali Kaafar, Steve Uhlig, and Johanna Amann, editors, *Passive and Active Measurement*, pages 3–16, Cham, 2017. Springer International Publishing.

[22] Samuel Schüppen, Dominik Teubert, Patrick Herrmann, and Ulrike Meyer. FANCI : Feature-based automated nxdomain classification and intelligence. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1165–1181, Baltimore, MD, August 2018. USENIX Association.

[23] Will Scott, Thomas Anderson, Tadayoshi Kohno, and Arvind Krishnamurthy. Satellite: Joint analysis of CDNs and network-level interference. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 195–208, Denver, CO, June 2016. USENIX Association.

[24] Dan Wing and Andrew Yourtchenko. Happy eyeballs: Success with dual-stack hosts. Technical report, 2012.

[25] Sandeep Yadav and A. L. Narasimha Reddy. Winning with DNS failures: Strategies for faster botnet detection. In Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis, editors, *Security and Privacy in Communication Networks*, pages 446–459, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[26] Sebastian Zander and Xuequn Wang. Are we there yet? IPv6 in Australia and China. *ACM Transactions on Internet Technology*, 18(3), February 2018.

A Decentralized Blockchain with High Throughput and Fast Confirmation

Chenxing Li*, Peilun Li*, Dong Zhou, Zhe Yang†, Ming Wu†,
Guang Yang†, Wei Xu, Fan Long‡†, Andrew Chi-Chih Yao
Tsinghua University †*Conflux Foundation* ‡*University of Toronto*

Abstract

This paper presents Conflux, a scalable and decentralized blockchain system with high throughput and fast confirmation. Conflux operates with a novel consensus protocol which optimistically processes concurrent blocks without discarding any as forks and adaptively assigns weights to blocks based on their topologies in the Conflux ledger structure (called Tree-Graph). The adaptive weight mechanism enables Conflux to detect and thwart liveness attack by automatically switching between an optimistic strategy for fast confirmation in normal scenarios and a conservative strategy to ensure consensus progress during liveness attacks.

We evaluated Conflux on Amazon EC2 clusters with up to 12k full nodes. The consensus protocol of Conflux achieves a block throughput of 9.6Mbps with 20Mbps network bandwidth limit per node. On a combined workload of payment transactions and Ethereum history transactions, the end-to-end system of Conflux achieves the throughput of up to 3480 transactions per second while confirming transactions under one minute.

1 Introduction

Following the success of cryptocurrencies [2, 23], blockchain has evolved into a technology powering secure, decentralized, and consistent transaction ledgers at Internet-scale. Newer blockchain platforms such as Ethereum [2, 35] support customized transaction rules as smart contracts, which greatly extend the capability of blockchain ledgers beyond value transfers.

Blockchain platforms like Bitcoin [23] use *Nakamoto consensus*. It organizes transactions into an ordered list of blocks, each of which contains multiple transactions and a link to its predecessor. Participants (miners) solves *proof-of-work* (PoW) puzzles to compete for the right of generating the next block. To prevent an attacker from reverting previous transactions, honest participants agree on the longest chain of blocks as the correct history. Each new block is appended at the end of the longest chain to make the chain longer and therefore harder to revert.

However, the performance remains one of the most critical issues of blockchains. Nakamoto consensus is bottlenecked by its slow block generation rate. For example, Bitcoin generates one 1MB block every 10 minutes and can therefore only process 7 transactions per second. Users have to wait for typically one hour (i.e., six blocks) to obtain high confidence on the finality of a transaction.

An ideal blockchain has the following four desirable properties, *security*, *decentralization*, *high throughput*, and *fast confirmation*. The key challenge of building such a blockchain system is the threat of security attacks. To obtain high performance, the system typically has to operate with a fast block generation rate. Because block propagation takes time, the system may therefore generate many concurrent blocks (i.e., forks). In Nakamoto consensus, concurrent blocks waste PoW computation because they do not contribute to the finality of the longest chain. They make the system vulnerable to *double spending attacks* that attempt to revert history transactions.

Moreover, a high block generation rate can make several recently proposed protocols vulnerable to *liveness attacks* [17, 31, 32]. An attacker can simultaneously generate blocks at two competing branches and strategically withhold/release these blocks to maintain the balance of the two branches. The attacker with little PoW computation power can stall the consensus progress [36].

Conflux: We present Conflux, the first blockchain system that achieves all of the four desirable properties. Conflux can process thousands of transactions per second while confirming each transaction with within one minute on average. With its novel consensus protocol, the consensus layer of Conflux is no longer the performance bottleneck, i.e., the throughput saturates its underlying gossip network bandwidth and the confirmation speed is within the same order of magnitude as the gossip network propa-

*The first two authors contributed equally.

gation delay. Conflux is provably secure (see our formal proof in [19]). It is also as decentralized and permissionless as Bitcoin — participants can join and leave the consensus process at any time and there is no privileged committee or super-node dictating the process. Conflux also implements a modified version of Ethereum Virtual Machine (EVM) [35] and most smart contracts in Ethereum can be directly ported to Conflux.

To address the security attack challenge, Conflux organizes blocks into a novel Tree-Graph structure, which is a tree embedded inside a direct acyclic graph (DAG). In Tree-Graph, concurrent blocks are not considered harmful and they contribute to the Conflux ledger as well. Their PoW solutions will improve the finality of all of their ancestors and their transactions will be optimistically included into the ledger total order. This secures Conflux against double spending attacks and improves the Conflux throughput. To address liveness attacks, the consensus protocol of Conflux inherently encodes two different block generation strategies: an optimistic strategy that allows fast confirmation and a conservative strategy that ensures the consensus progress. Conflux uses its novel *adaptive weight* mechanism to combine these two strategies into a unified consensus protocol.

Adaptive Weight: Conflux assigns a weight to each block, which indicates the amount of finality that the block contributes to its ancestors. For each new block, Conflux analyzes its topology in the Tree-Graph, decides whether a liveness attack is potentially going on (e.g., whether there are old ancestor blocks that are not finalized yet), and then adaptively assigns weights to blocks in the Tree-Graph to switch between the two strategies. In normal scenarios, the mechanism assigns weights in one way that enables the optimistic strategy to confirm transactions fast. When a liveness attack happens, the mechanism assigns weights in another way that enables the conservative strategy to thwart the attack.

Deferred Execution: The execution order of recently packaged transactions may oscillate temporarily in a system with fast block generation. A naive implementation of the transaction execution engine would have to roll back executions many times and waste computation resources. Conflux addresses this challenge with its *deferred execution* mechanism. Instead of executing transactions in every received block immediately, Conflux waits for several blocks so that the order is relatively stabilized. Our observation is that users need to wait for the stabilization of the total order anyway to confirm a transaction with high confidence. Therefore the deferred execution does not harm the user experience at all.

Link-Cut Tree: An efficient consensus implementation is important to the performance of Conflux. To maintain the Conflux Tree-Graph, a naive implementation has the time cost of $O(n)$ for processing a new block on average, where n is the number of existing blocks. To address this challenge, Conflux uses *link-cut tree* to maintain weight values in Tree-Graph efficiently. It reduces the processing time from $O(n)$ to $O(\log n)$ per block.

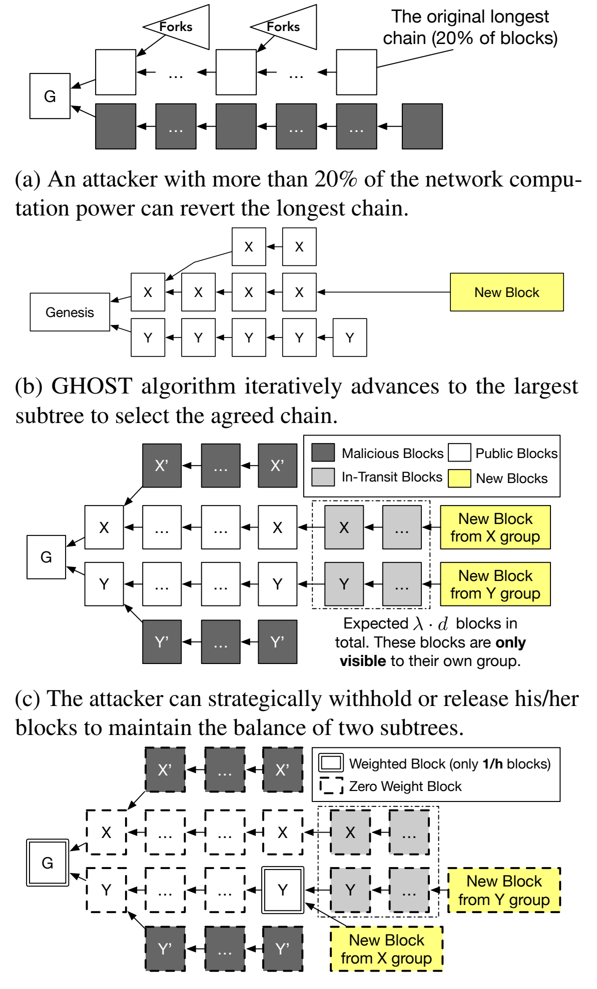
Experimental Results: We implemented Conflux and evaluated it with Amazon EC2 machines under the same experimental setup as previous work like Algorand and OHIE [11, 36]. Our experimental results show that with the bandwidth limit of 20Mbps and the simulated real world network latency setting, Conflux achieves a transaction throughput of 9.6Mbps and a confirmation latency of 47.75-51.54 seconds when running 3000-12000 nodes. For a combined workload of payment transactions and Ethereum history transactions, Conflux achieves up to 3480 transactions per second and confirms transactions within one minute with high confidence. Comparing to Algorand [11], Conflux has more than 4x higher throughput and comparable confirmation speed. Comparing to OHIE [36], Conflux has the same throughput and one order of magnitude faster confirmation speed.

Contribution: This paper makes the following contributions: 1) we design and implement Conflux, a decentralized and smart-contract-enabled blockchain system with high throughput and fast confirmation; 2) we present a novel consensus protocol with the adaptive weight mechanism; 3) we present a set of novel and critical optimizations, including deferred execution and link-cut tree.

2 Related Work

Nakamoto Consensus in Bitcoin: Transactions are packed into blocks in Bitcoin. Each block has one predecessor block and all blocks form a tree structure with the genesis block as the root. Participants agree on the longest chain as the valid transaction history. Nakamoto consensus has to use a relatively slow block generation rate to avoid the generation of concurrent blocks, i.e., forks. This is essential for the safety against double spending attacks as shown in Figure 1a. More forks would mean relatively less blocks in the longest chain. In Figure 1a, due to forks, only 20% of blocks are on the longest chain so that an attacker with more than 20% of the network computation power can revert the longest chain to launch double spending attacks.

GHOST: GHOST is a previous proposal to replace the longest chain rule to improve the consensus safety under a fast block generation rate [32]. It is partially implemented in Ethereum [2]. Figure 1b presents an exam-


Forks
Forks
The original longest chain (20% of blocks)
G
...
X
Genesis
Y
New Block
X'
Malicious Blocks
Public Blocks
In-Transit Blocks
New Blocks
New Block from X group
New Block from Y group
Expected $\lambda \cdot d$ blocks in total. These blocks are **only visible** to their own group.
Y'
Weighted Block (only **1/h** blocks)
Zero Weight Block

(a) An attacker with more than 20% of the network computation power can revert the longest chain.

(b) GHOST algorithm iteratively advances to the largest subtree to select the agreed chain.

(c) The attacker can strategically withhold or release his/her blocks to maintain the balance of two subtrees.

(d) If *h* is large enough, two groups will converge eventually.

Figure 1: Double Spending Attack, GHOST, Liveness Attack, and Structured GHOST

ple to illustrate the GHOST algorithm. GHOST starts from the genesis block and iteratively advances to the child block with the largest subtree to select the agreed chain [32]. In Figure 1b, the new block appends to the end of the agreed chain, which is in the subtree of X (containing 6 blocks) not in the subtree Y (containing 5 blocks). The difference between GHOST and the longest chain rule is that all blocks generated by honest participants will contribute to the finality of the agreed chain. Suppose G is an old enough block that is on the agreed chains of all honest participants. Future blocks generated by honest participants will all contribute the finality of G regardless of whether they are concurrent or not, because all of these blocks will be under the subtree of G. Unlike the longest chain rule, an attacker would need more than half of computation power to revert *G* from the agreed chain even with the presence of concurrent blocks [32].

Liveness Attack on GHOST: Unfortunately, GHOST is vulnerable to liveness attacks if the block generation rate is very fast. Figure 1c presents one example of such attacks. The example has the following settings: 1) the total block generation rate of honest participants is λ; 2) honest participants are devided into two groups with equal computation power (group X and group Y in Figure 1c); 3) blocks will transmit instantly inside each group, but the propagation between these two groups has a delay of d. In Figure 1c, each of the two groups extends its own subtree following the GHOST rule. Note that recent generated blocks within the time period of d are in-transit blocks (gray blocks in Figure 1c), which are only visible by the group who generates them. Therefore each group will believe its own subtree is larger until one group generates sufficiently more blocks than the other to overcome the margin caused by the in-transit blocks.

In normal scenarios, one of the two groups will get lucky to enable the blockchain to converge. However, an attacker can mine under two subtrees simultaneously to delay the convergence. The attacker can strategically withhold or release the mined blocks to maintain the balance of the two subtrees as shown in Figure 1c.

Theoretically, if honest participants evenly split due to network delay and the margin caused by in-transit blocks is significant, i.e., $\lambda d > 1$, a small portion of computation power is enough to launch attacks. Previous work [36] includes a simulation shows only 10% will do. In practice, even if honest participants do not have an even partition, the more computation power the attacker controls, the more likely the attacker will succeed. Consider the presence of mining pools, it is not rare to see one miner controlling more than 20% of computation power. Such a miner will be able to launch successful balance attacks without even partition.

DAG-based Structures: To improve the throughput and the confirmation speed, researchers have explored several alternative structures to organize blocks. Inclusive blockchain [17] extends the Nakamoto consensus and GHOST to DAG and specifies a framework to include off-chain transactions. In PHANTOM [31], participating nodes first find an approximate k-cluster solution for its local block DAG to prune potentially malicious blocks. They then obtain a total order via a topological sort of the remaining blocks. Unfortunately, when the block generation rate is high, inclusive blockchain and PHANTOM are all vulnerable to liveness attacks similar to Figure 1c. Therefore, unlike Conflux they cannot achieve both the security and the high performance.

Some protocols attempt to obtain partial orders instead of total orders for payment transactions. SPECTRE [30] produces a non-transitive partial order for all pairs of blocks in the DAG. Avalanche [4] connects raw transactions into a DAG and uses an iterative random sampling algorithm to determine the acceptance of each transaction. Unlike Conflux, it is very difficult to support smart contracts on these protocols without total orders.

Hierarchical and Parallel Chains: Besides DAG, alternative ways to organize blocks include hierarchical chains and parallel chains. For example, in BitcoinNG [9], a macro block is generated every 10 minutes. The miner of such a block becomes the leader to generate micro blocks that contain actual transactions until the next macro block. Similarly, FruitChain [27] packs transactions first into fruits (i.e., micro blocks) and then packs fruits into blocks. OHIE [36] runs multiple parallel chains with the standard Nakamoto consensus and then deterministically sorts blocks to obtain a total order.

The shared property of these protocols is that only a small portion of blocks (e.g., macro blocks in BitcoinNG and FruitChain) influence the total order of the transaction ledger. It mitigates the liveness attack issue in Figure 1c because it reduces the chance of in-transit blocks influencing the total order. But these protocols have slow confirmation speed because they need to wait for more blocks to confirm transactions than other protocols. For example, BitcionNG has the same slow confirmation speed as Bitcoin [9]; OHIE confirms transactions in about 10 minutes on average only under an extremely fast block generation rate of 64 blocks per second [36]. In contrast, Conflux has a much faster confirmation speed in normal circumstances with no ongoing liveness attack.

Prism operates with one proposer chain and many parallel vote chains, each of which casts vote to decide the total order of blocks in the proposer chain [6]. The theoretical simulation in [6] shows that Prism may achieve high throughput and fast confirmation speed similar to our Conflux results in Section 6. But the simulation assumes a block propagation delay of one second, which is too ideal (e.g., the measured block delay is 10-15 seconds in our experiments). It is therefore unclear how fast a blockchain system that implements Prism can confirm transactions when running under practical P2P networks.

Byzantine Fault Tolerance: ByzCoin [14] and Thunderella [28] propose to achieve consensus by combining the Nakamoto consensus with Byzantine fault tolerance (BFT) protocols. Algorand [11], HoneyBadger [22], and Stellar [21] replace the Nakamoto consensus entirely with BFT protocols. In practice, all these proposals run BFT protocols within a confined group of nodes, since BFT protocols only scale up to dozens of nodes. The confined group is often chosen based on their recent PoW computation power [14, 28], their stakes of the system [11], or external hierarchy of trusts [21, 22]. However, these approaches may create undesirable hierarchies among participants and compromise the decentralization of blockchain systems. In contrast, Conflux allows any participant to join and leave the network without permission. In addition, instead of *eagerly* deciding the total order of blocks as in BFT-based approaches, Conflux allows multiple blocks to be generated in parallel and finalizes their orders later, which leads to its higher throughput.

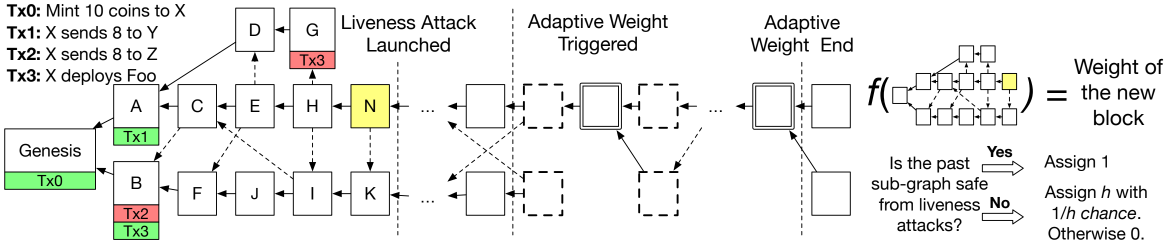
Sharding: Elastico [20], OmniLedger [15], RapidChain [37], and Monoxide [33] split the blockchain state into shards. Instead of having every node to verify all transactions, the systems select a small committee to maintain each shard to improve scalability. Unlike Conflux, such systems sacrifice security for scalability, i.e., the committee configuration can only be changed slowly (like days) due to reconfiguration overhead and therefore a small shard may be vulnerable to powerful attackers who can adaptively corrupt participants. This security issue is so important that Vault [16] chooses to only use sharding to mitigate storage cost with the trade-off of increased network bandwidth cost. Also despite the high combined throughput of all shards, the throughput of inter-shard transactions is still limited.

3 Overview

We will first present an example to illustrate a straw-man algorithm called *structured GHOST* that can defend against liveness attacks but has a sub-optimal confirmation speed. We will then present an overview of the Conflux consensus protocol, which uses the straw-man algorithm as a building block.

Structured GHOST: In our structured GHOST algorithm, only $1/h$ of blocks are weighted blocks that would count in the chain selection process. These blocks are selected randomly based on their PoW solution qualities (e.g., the number of leading zeros of the PoW hash). During the chain selection, the structured GHOST iteratively advances to the subtree with the largest number of weighted blocks, instead of considering all blocks.

Figure 1d shows an example to illustrate how structured GHOST can defend against the liveness attack we described before. With a sufficiently large h value, the expected number of in-transit weighted blocks ($\lambda d/h$) will be very small. As shown in Figure 1d, the generation of a weighted block of one group will very likely to make the two subtrees to converge, unless another group generates a concurrent weighted block. When $\lambda d/h \ll 1$, the chance of such concurrent generation is very unlikely. Without the margin caused by the in-transit blocks, the



(a) Tree-Graph Example. Solid blocks have weight one. Double-line blocks have weight 600. Dotted-line blocks have weight zero.

(b) Adaptive Weight

Figure 2: Examples of Conflux Consensus on Tree-Graph

liveness attack is not possible without significant computation power. Although structured GHOST is secure against liveness attacks, it sacrifices the confirmation speed — a user has to wait for the accumulation of enough weighted blocks to confirm a transaction.

Consensus with Two Strategies: The Conflux consensus protocol operates with two strategies, an optimistic strategy similar to the GHOST algorithm and a conservative strategy similar to the above straw-man algorihtm. Our adaptive weight mechanism enables Conflux to encode these two strategies in a unified framework. In normal scenarios, Conflux would use the optimistic strategy to achieve high performance. If a liveness attack happens, the adaptive weight mechanism enables honest participants of Conflux to cooperatively switch to the conservative strategy to thwart the attack automatically.

Tree-Graph: The Conflux consensus protocol operates on the local Tree-Graph state of each individual node. Figure 2a presents a running example of the local Tree-Graph state of a node in Conflux. We will use this example in the remaining of this section to illustrate the high-level ideas of the Conflux consensus protocol. Each vertex in the Tree-Graph in Figure 2a corresponds to a block. In Figure 2a, Genesis is the predefined genesis block. Only Genesis, A, B, and G are associated with transactions. There are two kinds of edges in the Tree-Graph, parent edges and reference edges:

Parent and Reference Edges: Each block except Genesis has exactly one outgoing parent edge (solid line arrows in Figure 2a). For example, there is a parent edge from C to A. Each block can have multiple outgoing reference edges (dashed line arrows in Figure 2a). A reference edge corresponds to generated-before relationships between blocks. For example, there is an edge from E to D. It indicates that D is generated before E.

Pivot Chain: Note that all parent edges in the Tree-Graph together form a *parental tree* in which the genesis block is the root. In the tree, Conflux selects a chain from the genesis block to one of the leaf blocks as the *pivot chain*. Each block in the Tree-Graph may have a different weight determined by our novel adaptive weight mechanism. Conflux iteratively advances to the subtree with the heaviest total block weight to select the pivot chain. In Figure 2a before the liveness attack, Conflux selects Genesis, A, C, E, and H as the pivot chain to append the new block N. Note that Conflux does not selects the chain of Genesis, B, F, J, I, and K, because the subtree of A has heavier weights than the subtree of B.

Generating New Block: Whenever a node generates a new block, it first computes the pivot chain in its local Tree-Graph state and sets the last block in the chain as the parent of the new block. The node then finds all tip blocks in the Tree-Graph that have no incoming edge and creates reference edges from the new block to each of those tip blocks. For example, in Figure 2a, when generating N, the node chooses H as the parent of N and creates a reference edge from N to K.

Adaptive Weight: Figure 2b illustrates the basic idea of the adaptive weight mechanism. The goal is to assign a different weight to each generated block so that Conflux can adaptively switch between the optimistic strategy with a fast confirmation and the conservative strategy that ensures the consensus progress. Conflux determines the weight of a new block based on its past sub-graph, i.e., all blocks that are reachable via a traversal from the new block. As shown in Figure 2b, if the past sub-graph is safe — every old enough ancestor of the new block in the past sub-graph is secured on the pivot chain with high probability, the weight of the new block will be one. If not, the new block will be assigned an adaptive weight — it gets a weight of h with the chance of $1/h$ (depending on the PoW quality) or zero otherwise. Note that we set $h = 600$ in Conflux. See Section 4.1.

Liveness Attack Resilience: Figure 2a shows how the adaptive weight mechanism stops liveness attacks. Suppose after the generation of N, an attacker launches a liveness attack similar to one described in Figure 1c to balance the subtree of A and B. After a while, all honest participants start to generate blocks with adaptive weights, because they find that the old ancestors of their new generated blocks (e.g., A or B in Figure 2a) are still not secured on the pivot chain with high probability. This essentially enables the consensus protocol to operate in the conservative strategy similar to the structured GHOST algorithm. In Figure 2a, the heavy weight blocks make Conflux to converge to the the subtree of A. After more blocks being generated under A, the past sub-graph of new generated blocks will become safe. Participants therefore assign weight one to the new blocks, automatically swtiching back to the optimistic strategy.

Epoch and Block Order: Parent edges, reference edges, and the pivot chain together enable Conflux to split all blocks in a Tree-Graph into epochs. Every block on the pivot chain corresponds to one epoch. Each epoch contains all blocks 1) that are reachable from the corresponding block in the pivot chain via the combination of parent and reference edges (including the pivot chain block itself) and 2) that are not included in previous epochs. For example, in Figure 2a, J belongs to the epoch of H because J is reachable from H but not reachable from the previous pivot chain blocks.

Conflux then derives a total order of all blocks in the Tree-Graph with the following rules. Conflux first sorts blocks based on their epochs. For blocks within the same epoch, Conflux sorts them based on their topological order. Conflux break ties determinisitcally (e.g., with the PoW quality or the block hash). For example, in Figure 2a, Conflux will obtain the following block total order for all blocks before the liveness attack: Genesis, A, B, C, D, F, E, G, J, I, H, K, and N.

Transaction Order: Conflux first sorts transactions based on the total orders of their enclosing blocks. If two transactions belong to the same block, Conflux sorts the two transactions based on the appearance order in the block. Conflux checks the conflicts of the transactions at the same time when deriving the order. If two transactions are conflicting with each other, Conflux will discard the second one. If one transaction appears in multiple blocks, Conflux will only keep the first appearance and discard all redundant ones. In Figure 2a, the transaction total order is Tx0, Tx1, ~~Tx2~~, Tx3, and ~~Tx3~~. Conflux discards Tx2 because it conflicts with Tx1.

4 Consensus on Tree-Graph

The local state of a node in Conflux is $S = \langle B, g \rangle$, where B is the set of blocks and $g \in B$ is the genesis block. There are several fields associated with a block $b \in B$. $b.\mathsf{parent}$ denotes the parent block of b. $b.\mathsf{pred\_blocks}$ denotes the set of predecessor blocks linked by the reference and parent edges from b. $b.\mathsf{pow\_quality}$ is the quality of the PoW solution — for b to be valid, $b.\mathsf{pow\_quality}$ must no less than the PoW difficulty D. $b.\mathsf{weight}$ is the adaptive weight of b. We use $b.\mathsf{hash}$ to denote the hash of b – all nodes in Conflux share a predefined deterministic hash function that maps each block to a unique id.

$$\mathsf{Child}(B,b) = \{b' \mid b' \in B,\, b'.\mathsf{parent} = b\}$$
$$\mathsf{SubT}(B,b) = (\cup_{i \in \mathsf{Child}(B,b)} \mathsf{SubT}(B,i)) \cup \{b\}$$
$$\mathsf{SubTW}(B,b) = \textstyle\sum_{i \in \mathsf{SubT}(B,b)} i.\mathsf{weight}$$
$$\mathsf{Past}(b) = (\cup_{i \in b.\mathsf{pred\_blocks}} \mathsf{Past}(i)) \cup b.\mathsf{pred\_blocks}$$
$$\mathsf{PastW}(b) = \textstyle\sum_{i \in \mathsf{Past}(b)} i.\mathsf{weight}$$

Figure 3: The definitions of utility functions.

```
Input  : A set of blocks $B$ and a starting block $b$.
Output : The pivot block for the subtree of $b$.
1 if $\mathsf{Child}(B,b) = \emptyset$ then
2     return $b$
3 else
4     $w \leftarrow \max\{\mathsf{SubTW}(B,i) \mid i \in \mathsf{Child}(B,b)\}$
5     $a \leftarrow \operatorname{arg\,min}_{i \in \mathsf{Child}(B,b)} \{i.\mathsf{hash} \mid \mathsf{SubTW}(B,i) = w\}$
6     return $\mathsf{Pivot}(B,a)$
```

Figure 4: The definition of $\mathsf{Pivot}(B,b)$.

Figure 3 defines several utility functions and notations. $\mathsf{Child}()$ returns the set of child blocks of a given block. $\mathsf{SubT}()$ returns the set of blocks in the subtree of a given block in the parental tree. $\mathsf{SubTW}()$ returns the sum of the weights in the subtree. $\mathsf{Past}()$ returns the set of blocks that are generated before a given block. $\mathsf{PastW}()$ returns the sum of the weights of the past block set of a block. Note that $\mathsf{Past}(b)$ and $\mathsf{PastW}(b)$ are determined at the generation time of b and remain constant afterwards. In this section, we use lists to denote chains and serialized orders. "$\circ$" denotes the concatenation of two lists.

4.1 Pivot Chain and Adaptive Weight

Pivot Chain: Figure 4 presents the pivot chain selection algorithm in Conflux. Given a set of blocks B and the starting genesis block g, $\mathsf{Pivot}(B,g)$ returns the leaf block of the selected pivot chain. The algorithm recursively advances to the child block whose corresponding subtree has the largest total weights (lines 4-6). To break ties, the algorithm selects the child block with the smallest unique hash id (line 5). The algorithm terminates until it reaches a leaf block (lines 1-2).

Adaptive Weight: Figure 5 presents how we calculate the weight of a block b. The algorithm first determines whether the block should have adaptive weight or not based on the past block set of b (lines 1-11). If not, the weight of the block will be one (lines 12-13). If so, the algorithm checks the PoW solution quality against a difficulty that is h times higher than the base validation difficulty. The weight of the block will be h if it passes the check and be zero if it fails (lines 14-17).

To determine whether b should have adaptive weight, the algorithm operates at a sub-Tree-Graph that only contains blocks in the past set of b. It inspects every block in

```
Input   : A new block $b$
Output  : The adaptive weight of $b$
1  $B \leftarrow \mathsf{Past}(b)$
2  $a \leftarrow b.\mathsf{parent}$
3  $adaptive \leftarrow$ False
4  Let $f(x) =$
     $2 \cdot \mathsf{SubTW}(B,x) - \mathsf{SubTW}(B,x.\mathsf{parent}) + x.\mathsf{parent.weight}$
5  Let $t(x) = |\mathsf{TimerChain}(b)| - |\mathsf{TimerChain}(x.\mathsf{parent})|$
6  Let $g(x) = |\mathsf{SubT}(B,x.\mathsf{parent})|$
7  while $a.\mathsf{parent} \neq$ Nil do
8      if $f(a) < \alpha$ and $(t(a) > \beta$ or $g(a) > \gamma)$ then
9          $adaptive \leftarrow$ True
10         break
11     $a \leftarrow a.\mathsf{parent}$
12 if not $adaptive$ then
13     return 1
14 else if $b.\mathsf{pow\_quality} \geq h \cdot D$ then
15     return $h$
16 else
17     return 0
```

Figure 5: The definition of $\mathsf{AdaptiveWeight}(b)$

```
Input   : A block $b$.
Output  : The timer chain of the past sub-graph of $b$.
if $b.\mathsf{pred\_blocks} = \emptyset$ then
    return $b$
else
    $a \leftarrow \underset{i \in b.\mathsf{pred\_blocks}}{\arg\max} \{|\mathsf{TimerChain}(i)|\}$
    if $b.\mathsf{pow\_quality} > h_0 \cdot D$ then
        return $\mathsf{TimerChain}(a) \circ b$
    else
        return $\mathsf{TimerChain}(a)$
```

Figure 6: The definition of $\mathsf{TimerChain}(b)$.

the path from the genesis to $b.\mathsf{parent}$. For each inspected block a, it determines 1) whether a is still not secure on the pivot chain with high probability – the subtree weight of a is not significantly larger than the weight of the sibling subtrees of a (i.e., $f(a) < \alpha$) and 2) whether a is old enough – there is an enough amount of timer ticks or an enough number of blocks in the subtree of its parent (i.e., $t(a) > \beta$ or $g(a) > \gamma$). If any inspected block satisfies these two conditions, b should have adaptive weight.

The intuition is that to make progress, for any pivot chain block a' in Tree-Graph, after a certain period of time, one of the child subtree of a' (e.g., the subtree of a) should become dominant. If the attacker attempts to maintain the balance between the subtrees of two (or more) children of a' for a long time, the condition at line 8 will become true for a. Therefore, all honest participants will start to generate blocks with adaptive weights. Conflux will essentially operate with a conservative strategy similar to the structured GHOST algorithm (see Section 3). This will thwart the attack to ensure the progress.

Timer Chain: Because an attacker with enough computation power may influence the subtree sizes of recent pivot chain blocks via strategically withholding mined blocks, only counting the number of blocks under the subtree of a pivot block is not sufficient to detect whether the pivot block is old enough or not. To this end, Conflux uses a timer chain mechanism to obtain an attacker resilient estimation for the generation time of each block.

```
Input   : The local state $S = \langle B, g \rangle$ and a new discovered
          block $b$
1  if $b.\mathsf{pow\_quality} \geq D$ then
2      Wait until $\mathsf{Past}(b) \subseteq B$
3      if $\mathsf{Pivot}(\mathsf{Past}(b), g) = b.\mathsf{parent}$ then
4          $b.\mathsf{weight} \leftarrow \mathsf{AdaptiveWeight}(b)$
5          $S \leftarrow \langle B \cup \{b\}, g \rangle$
```

Figure 7: The block validation procedure.

```
Input   : A block $b$
Output  : An ordered list of all blocks in Past($b$) $\cup \{b\}$
if $b$.parent = Nil then
    return $\emptyset$
$L \leftarrow$ ConfluxOrder($b$.parent)
$B_\Delta \leftarrow$ (Past($b$) $-$ Past($b$.parent) $- \{b$.parent$\}) \cup \{b\}$
while $B_\Delta \neq \emptyset$ do
    $B'_\Delta \leftarrow \{x \mid |x.\mathsf{pred\_blocks} \cap B_\Delta| = 0\}$
    Sort all blocks in $B'_\Delta$ in order as $a_1, a_2, \ldots, a_k$
      such that $\forall 1 \leq i < j \leq k$, $a_i.\mathsf{hash} < a_j.\mathsf{hash}$
    $L \leftarrow L \circ a_1 \circ a_2 \circ \ldots \circ a_k$
    $B_\Delta \leftarrow B_\Delta - B'_\Delta$
return $L$
```

Figure 8: The definition of ConfluxOrder().

Figure 6 presents the definition of $\mathsf{TimerChain}(b)$, which is the longest chain of blocks in the past sub-graph of b whose PoW qualities are h_0 times higher than the normal difficulty. We then use the length of the timer chain as the timer tick of the generation time estimation of each block (i.e., line 5 in Figure 5). When h_0 is large enough respecting the network delay, the attacker cannot stop the growth of the timer chain, because honest participants will contribute to the timer chain almost synchronously. In Conflux we set $h_0 = 360$. See Section 6.1.

4.2 Block Validation and Total Order

Block Validation: Figure 7 presents the validation procedure for a new discovered block. It first checks whether the block has a PoW solution with a sufficient quality (line 1). The procedure will wait for all blocks in its past sub-graph being processed first (line 2). The procedure will then compute the pivot chain in its past sub-Tree-Graph to check whether it selects the right parent (line 3). If so, the procedure computes the weight of the new block and adds it to the local Tree-Graph state (lines 4-5).

Block Order: Figure 8 defines $\mathsf{ConfluxOrder}()$, our block ordering algorithm. Given a block b, $\mathsf{ConfluxOrder}(b)$ returns the total order of all blocks in $\mathsf{Past}(b) \cup \{b\}$. The algorithm sorts the blocks based on their corresponding epochs, i.e., it first recursively orders all blocks in previous epochs. It then computes all blocks in the epoch of b as B_Δ (line 4). It topologically sorts all blocks in B_Δ and appends them to the result list (lines 5-10), and uses the hashes to break ties (lines 7-8).

4.3 Correctness

We next discuss the intuitions behind the correctness of our consensus algorithm. Suppose the network together has a block generation rate of λ. The correctness of Conflux is based on the following assumptions: 1) attackers control at most δ of the total block generation power ($\delta < 0.5$); 2) the network is d-synchronous, i.e., if at time t one honest node broadcast a block via the gossip network, then before time $t+d$, all honest nodes will receive this block and add this block into their local states.

Adversary Model: The attacker can choose arbitrary strategies to disrupt honest nodes. We also assume 1) attackers immediately receive all blocks and transactions from the gossip network, 2) attackers can arbitrarily control the communication of honest nodes as long as the d-synchronous assumption holds. The attacker however does not have the capability to reverse cryptographic functions. Therefore honest nodes can reliably verify the integrity of a block in the presence of the attacker.

Safety: Conflux is safe against double spending attacks because of two facts: 1) to revert a transaction in an epoch, the attacker has to revert the pivot chain block associated with the epoch from the pivot chain; 2) reverting an old pivot chain block that is on the common pivot chain of all honest nodes requires the attacker to compete with all honest nodes together. Although honest nodes may generate blocks that are concurrent with each other, all of these blocks will be under the subtree of the common pivot chain block. As the time passes by, it will be impossible for the attacker to forge an alternative heavier subtree without the pivot chain block.

Liveness: Many previous consensus algorithms based on tree and DAG can only provide liveness guarantees if the block generation rate is significantly slower than the block propagation delay (i.e., $\lambda \cdot d \ll 1$) [13, 26]. In contrast, Conflux is safe against liveness attacks at the protocol level even when the block generation rate is fast, because if the consensus does not make progress for a certain period of time, all honest nodes will start to generate blocks with adaptive weights. In this scenario, only blocks with very high PoW quality will decide the total order and the block generation rate of these blocks is significantly slower than the block propagation delay (i.e., $\frac{\lambda \cdot d}{h} \ll 1$ for a large enough h). Because concurrent generation of such heavy weight blocks is rare, an attacker has to release a large number of previously withheld blocks to balance a new generated heavy block or all honest nodes will make progress and will recognize the heavy block as a common pivot chain block. Because block withholding capability of an attacker is limited by its block generation power, the attacker will eventually run out of blocks to continue the liveness attack. We prove in [19] that when $\delta < 1/2$, once a block in Conflux is seen by an honest node, its order will become irreversible with exception risk ε after $d \cdot O(\log(1/\varepsilon))$ time.

Confirmation Policy: Conflux confirms a block b if for any ancestor block of b, the corresponding subtree total weight is heavier than all of the subtrees of its siblings by a margin. This margin is not a preset value. It depends on the status of blockchain protocol. With the parameter setting used in our experiments, this margin is about 20∼30 in normal scenarios for obtaining the same confidence as waiting six blocks in Bitcoin. Specially, if Conflux is in the conservative mode and is generating blocks with adaptive weights, we need to wait six blocks with heavy weights instead. See [19] for the detailed formulas of the risk in confirming a block.

5 Implementation and Optimizations

We have implemented Conflux in Rust [1].

Difficulty Adjustment: For brevity, our algorithm in Section 4 assumes a constant PoW difficulty. Conflux operates with a difficulty adjustment mechanism tailored for Tree-Graph. Every 5000 epochs, Conflux counts the number of blocks generated in the last 5000 epochs and adjusts the difficulty accordingly to maintain a stable block generation interval. Instead of setting the weight of every normal block to one, Conflux sets the weight to the difficulty of the block. For a block with a heavy adaptive weight, its weight will be its difficulty multiplied by h. The rationale is to allow the block weight to align with the accumulated PoW as the difficulty changes.

Storage: A Conflux full node stores the blockchain account state as Merkle Patricia Trees [35] in a key-value DB. To save storage space, Conflux periodically forms checkpoints at specific epoch heights (e.g., every 200k epoch heights) when the confirmation risk of the pivot chain blocks at these heights become extremely low. After a checkpoint, all history transactions before it can be safely discarded — all full nodes treat the checkpoint block as the new genesis. Note that full nodes still store all block headers to help new nodes bootstrapping.

Bootstrap: To bootstrap a new full node to join the network, it first synchronizes all the block headers in the ledger from the peers and decides the latest confirmed checkpoint block based on the headers. It then fetches the corresponding checkpoint state from the peers and continues the execution from that state.

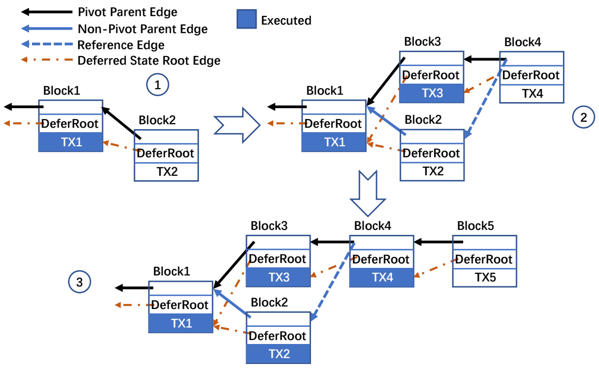


Figure 9: Save Redundant Execution by Deferred Execution

Transaction Relay: For a high performance blockchain like Conflux, it is critical to minimize the redundant transactions that are transferred. Ideally and optimally, each node should only receive each transaction exactly once. In current Bitcoin and Ethereum implementation, transactions are disseminated among nodes via flooding, which may waste network bandwidth resources. Erlay [24] tries to solve this issue of Bitcoin by letting peers exchange sets of unsent transactions that are encoded with PinSketch algorithm [8]. However, this method cannot be applied in Conflux, since it only works well when the difference between the transaction sets of two peers is small. Conflux is not this case because the transaction throughput of Conflux is orders of magnitude higher than Bitcoin. Conflux instead only floods 4-byte short transaction ids and pulls the missed transactions from peers. The short id is built by a SipHash [12] on the SHA-3 hash of the transaction and a peer-specific random nonce to significantly decrease the id conflict rate.

Signature Verification: Signature verification is computation-intensive and may become a bottleneck when the throughput is high. Conflux therefore uses a thread-pool to parallel the signature verification for different transactions to avoid bottlenecking other components.

Incentive Mechanism: For every mined block, Conflux assigns its block generation reward based on how many other blocks that are generated in parallel in Tree-Graph, i.e., blocks that are not in the past and future sets of the mined block. The more blocks are in parallel with the mined block, the smaller the block reward would be. This incentive mechanism penalizes malicious behaviors such as withholding mined blocks and not referencing other blocks. Because every block receives reward regardless of whether they are on the pivot chain or not, this mechanism nullifies selfish mining attack strategies [10, 25, 29]. See [7] for the details of the incentive mechanism.

5.1 Deferred Execution

In Conflux, when a block just enters the Tree-Graph structure, its position in the total order will change frequently. Although such oscillation will stop in a short time, it poses a challenge for the transaction execution engine. In typical blockchain systems, all transactions in a block immediately get executed in a node as soon as it is discovered. Such naive approach may execute transactions in a block many times as the order of the block oscillates. This incurs significant execution overhead. The top part of Figure 9 illustrates this problem. When a full node just gets the $Block_2$, it is on the pivot chain and the total order of transactions is $\{TX_1, TX_2\}$. TX_2 is then executed in this order. But, when the node later gets $Block_3$ and $Block_4$, $Block_2$ does not belong to pivot chain anymore and TX_3 is positioned between TX_1 and TX_2 in the newly decided total order. TX_2 has to be executed again.

Conflux uses a novel *deferred execution* mechanism to address this issue. The insight is that, just like users waiting for a period of time to confirm transactions, Conflux can wait for the total order position of a block to almost stabilize to execute its transactions. Conflux delays the execution by k-epochs. Specifically, unlike Ethereum where the header of each block b contains a merkle state root that corresponds to the execution results after processing all transactions in and before b, the header of b in Conflux contains a deferred root that corresponds to the execution results of the block that are k-hops older than b along its path to the root. We set k to five in Conflux so that Conflux can avoid re-execution of transactions in most cases. Five is also smaller than the typical number of epochs that an user needs to wait to confirm transactions and therefore it does not impact the user experience.

Figure 9 presents an example to illustrate the saving of redundant executions by using deferred execution. For illustration purpose, we set k to be one in this example. Therefore in Figure 9, the $Block_2$ stores the state root based on the execution of TX_1. When the full node gets $Block_3$ and $Block_4$, in order to verify the deferred state root of $Block_4$, the system needs to execute TX_3 but does not need to execute TX_2 because TX_2 is positioned after TX_3 in the decided total order. When getting and verifying $Block_5$, Conflux then needs to produce the state based on the execution of TX_4 which depends on TX_2. In the process, although the pivot chain oscillates between $Block_2$ and $Block_3$, TX_2 only gets executed once.

5.2 Link-cut Tree Optimizations

Maintaining pivot chain in Conflux is not trivial. Adding a new block to the Tree-Graph requires updating the subtree weights of all the blocks from this new block back to the genesis. The naive approach will takes $O(n)$ time to complete, because the Tree-Graph height is usually linear to the number of blocks. To efficiently update the subtree weights, Conflux uses a data structure called *link-cut tree* [3]. Link-cut tree splits a tree structure into one or more paths, and represents each path using a splay tree, a form of balanced binary search tree invented by Tarjan et al. [5]. The link-cut tree is ideal for maintaining values like the subtree weights in the Conflux consensus protocol, because it enables the following operations at an amortized cost of $O(\log n)$: 1) increase or decrease values of all nodes along a path in the tree by a given value; 2) find the minimum or maximum value among values of all nodes along a path; 3) find the least common ancestor (LCA) of two nodes in the tree.

Update Pivot Chain: Conflux tracks the last pivot chain block of the current Tree-Graph state. Conflux uses the link-cut tree to maintain the total subtree weights of each block. When Conflux discovers a new block b, it inserts b into the link-cut tree and increases the total subtree weights of all blocks along the path from b to the root (i.e., genesis) by $b.\mathsf{weight}$. Note that adding b may trigger a pivot chain change — instead of running the chain selection algorithm from the root, Conflux uses the link-cut tree to calculate the LCA of b and the current last pivot block p. If the weight of the subtree p belongs to is still heavier than the one b belongs to, no pivot chain update occurs. Otherwise, Conflux re-runs the selection algorithm from the LCA block. Because long range pivot chain reorganization is extremely rare, rerunning the algorithm from the LCA block is not expensive in practice.

6 Experimental Results

We next present a systematic evaluation of Conflux on its throughput, confirmation speed, and scalability. We also evaluate important design aspects of Conflux, e.g., the adaptive weight mechanism for defending against liveness attacks, the deferred execution for optimizing the transaction execution, as well as link-cut tree for optimizing the Tree-Graph maintainance.

We deployed Conflux on up to 800 Amazon EC2 m5.2xlarge virtual machines (VM), each of which has 8 cores and 1Gbps network throughput. By default, we run one Conflux full node in each VM. To model the network latency, we use the intercity latency measurements [34] and assign each VM to one of 20 major cities. We emulate the intercity delay by inserting artificial delays. For each full node, the gossip network of Conflux connects it to an average of 10 randomly selected peers.

When we measure the confirmation speed of a transaction, we count a transaction as confirmed if we can obtain the same confidence as empirically confirming a transaction in Bitcoin after waiting six Bitcoin blocks. In our experiments, unless otherwise noted, we limit the bandwidth of each full node to 20Mbps and we assign each full node with an equal block generation power.

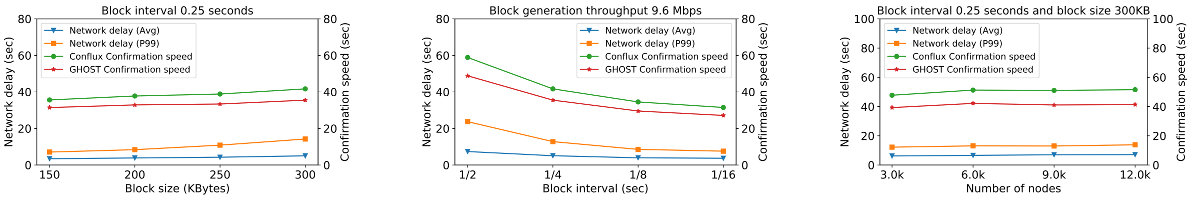
6.1 Protocol Parameter Calibration

To calibrate Conflux consensus protocol parameters, we run a set of experiments with 200 Conflux full nodes on Amazon EC2 with one full node per VM. We run Conflux with a set of different combinations of block size limits and block generation rates to measure the block propagation delays. For each setting, we run Conflux from the genesis for 10 minutes and fill each block to full with randomly generated simple payment transactions.

Figure 10a and 11a presents the experimental results of Conflux where we fix the block generation rate at four blocks per second and change the block size limit. Average network delay corresponds to the number of seconds on average for a generated block to reach more than 50% of participants. Network delay (99%) corresponds the number of seconds for all blocks to reach more than 99% of participants. In our experiments, we use the network delay (99%) number as the network diameter d for calculating parameters. There are often one or two machines lagging behind for some blocks and we can tolerate them as temporary failure nodes.

Conflux achieves the throughput of 9.6Mbps at the setting of 300K $\times$ 4 blocks per second. We find that Conflux almost saturates its underlying gossip network capability, considering that we limit the bandwidth of each full node to 20Mbps, which is only enough to send each block twice on average. With block sizes of 350K and beyond, full nodes start to experience significantly higher delay and may not be able to catch up new blocks.

Figure 10b and 11b presents the experimental results of Conflux where we fix the block generation throughput at 9.6Mbps and change the block generation rate from 2 blocks/s to 16 blocks/s. Our results show that as Conflux operates with faster block generation rate and smaller blocks, the network propagation delay decreases. But the delay no longer decreases much as it approaches the latency limit of the network. Smaller network propagation delay will improve the confirmation speed of transactions, but there are additional costs for using high generation rate. 1) Conflux full nodes have to store all block headers (block content could be pruned away with checkpoint techniques) and the average header size of Conflux is 300$\sim$500 bytes; 2) high block generation rates incur more blocks in parallel and these blocks cannot process transactions with dependencies.

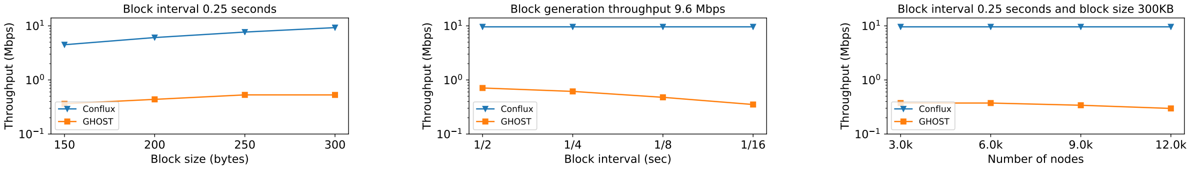


(a) Latency vs. block sizes.

(b) Latency vs. block intervals.

(c) Scalability

Figure 10: Network Delay and Confirmation Speed



(a) Throughput vs. block sizes.

(b) Throughput vs. block intervals.

(c) Scalability

Figure 11: Throughput

Based on the above trade-off, we choose the block generation rate of 4 blocks per second and the block size limit of 300K. With the measured network propagation delay, we determine the adaptive weight algorithm parameters following suggestions in our theory analysis [19]. 1) $h = 600$ is large enough to enable Conflux to tolerate liveness attacks from a powerful attacker that controls 40% of the network computation power; 2) $\beta = 160$ and $\gamma = 10000$ so that the confirmation policy gives a desirable margin; 3) $\alpha = 1800$ since it requires $\alpha \geq 3h$; 4) $h_0 = 360$ so the timer chain will have rare forks. Figure 10 plots the average confirmation speed under this set of parameters.

Compared to GHOST (by only considering pivot blocks as valid), Conflux achieves the similar confirmation latency while provides significantly higher throughput. As Figure 11b shows, the transaction throughput of GHOST decreases with increasing block generation rate since more concurrently generated blocks with smaller size lead to less valid transactions.

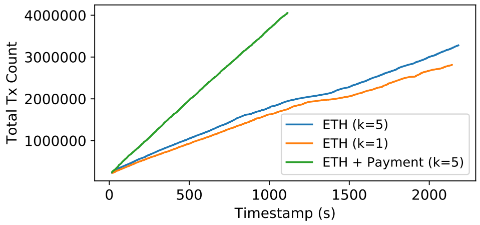


Figure 12: End-to-end Results for ETH Workload

6.2 Performance Results

Consensus Scalability Results: We next evaluate the consensus protocol performance as the number of nodes increases. Due to our resource limitation, we have to run 15 full nodes per VM. Because we are evaluating the consensus protocol only, we turn off signature verification and transaction execution to ensure enough computation resources for 15 full nodes sharing each VM.

Figure 10c presents the network propagation delay and the average transaction confirmation speed when running Conflux with different numbers of full nodes. In all the experiments, Conflux successfully operates with the block throughput of 9.6Mbps (300K × 4 blocks per second). Our results highlight the fast confirmation speed of Conflux, it confirms transactions on average in 47.75-51.54 seconds when running 3000-12000 full nodes. Our results also show that the consensus protocol of Conflux scales well. As the number of nodes increases, the network propagation delay only increases slightly so does the confirmation speed.

Note that our experimental setup is as same as the Algorand and OHIE papers [11, 36], therefore we can directly compare our results with their results. Compared to Algorand [11], Conflux achieves more than 4x throughput and similar confirmation latency. Compared to OHIE [36], Conflux achieves similar throughput but one order of magnitude faster confirmation.

End-to-end Results: With the calibrated parameters, we run Conflux on 400 VMs (one node per VM) to measure the throughput and the confirmation speed of Conflux. To obtain a representative workload, we collected the first four million transactions from the Ethereum blockchain [2]. This includes both payment transactions and smart contract transactions. We converted these col-

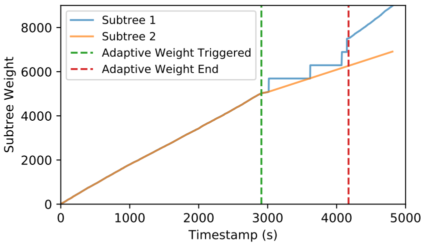


Figure 13: Subtree Weights under Liveness Attack

lected transactions into Conflux transaction format. We run two experiments, one experiment with the collected Ethereum transactions only and another experiment with a combined workload of the collected Ethereum transactions and randomly generated payment transactions. We terminate an experiment once Conflux processes four million transactions in total.

Figure 12 presents the number of processed transactions overtime. Our experimental results show that Conflux achieves a throughput of 1392 transactions per second for Ethereum workload and 3480 transactions per second for the combined workload. The average confirmation latencies are under one minute for both the Ethereum workload and the combined workload. Note that in the combined workload experiment, 14% of processed transactions are from Ethereum history.

Conflux achieves higher transaction throughput on the combined workload than on the Ethereum history workload. A primary reason is that Ethereum is a much slower blockchain and its transaction history does not have enough parallelism to saturates the Conflux consensus layer. We find that during the execution, future transactions in the Ethereum history often depends on previous transactions and full nodes of Conflux often do not have enough pending transactions ready to pick up so concurrent blocks pack duplicate transactions. A secondary reason is that Ethereum history contains more smart contract transactions which are more expensive to process than payment transactions.

Deferred Execution: Conflux by default defers the execution of transactions by five epochs $k = 5$. To illustrate the effect of the deferred execution optimization, we run a modified version of Conflux on the Ethereum history workload with $k = 1$. The results in Figure 12 show that this causes a 11.6% slowdown of Conflux on the transaction throughput because of the frequent re-execution of transactions during order oscillation.

6.3 Liveness Attack and Link-cut Tree

Liveness Attack: We conducted a liveness attack experiment to evaluate the security of Conflux. The experiment includes three nodes, two honest nodes and one attacker node. They keep a four block per second block generation rate for entire network. The block propagation network delay between the two honest nodes are 20 seconds and the attacker node does not relay honest blocks. We use only two honest nodes with significant delay to simulate the ideal liveness attack scenario in Figure 1c — a powerful attacker that evenly splits honest nodes in two groups and honest nodes with no latency inside a group and maximum latency between the two groups. The attacker node controls 30% of the total computation power. It launches the attack by finding the first fork between the two honest nodes and try to mine blocks under the lighter subtree to keep the two subtrees balanced.

Figure 13 shows how the weights of the two forked subtrees change along the time. The attack starts at the timestamp 0. During the time when the attack is performed and no adaptive weight is triggered, the weights of the two forked subtrees are almost perfectly balanced. The adaptive weight mechanism triggers the conservative strategy at timestamp 2,909 s. After that, the liveness attack quickly fails and the two honest nodes can then agree on the same pivot block and generate blocks under its subtree. After another 1,264 s, the two honest nodes come back to the optimistic strategy.

Link-cut Tree: To evaluate the benefits of link-cut tree, we run experiments on a micro-benchmark, a Tree-Graph that contains 1.5 million blocks, to measure the block processing throughput of the naive as well as our optimized approaches. Our experimental results show that the naive appraoch slows down to less than 4 blocks per second when the number of blocks in the Tree-Graph grows to one million, while our approach processes 5000 blocks per second on average.

7 Conclusion

Conflux is a scalable and decentralized blockchain platform with high throughput and fast confirmation. Its novel consensus protocol makes Conflux secure against both double spending attacks and liveness attacks, even if Conflux operates with a fast block generation rate. Conflux provides a promising solution to address the performance bottleneck of blockchains and opens up a wide range of blockchain applications.

Acknowledgments

We thank Bo Qiu and Yanpei Liu for their helps on the experiments. We thank Xi Wang and the anonymous reviewers for their insightful comments on the early draft of this paper. We note that [18] describes an early version of the Conflux system.

References

[1] Conflux-Rust. https://github.com/Conflux-Chain/conflux-rust.

[2] Etherum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Link/cut tree. https://en.wikipedia.org/wiki/Link/cut_tree.

[4] Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV.

[5] Splay tree. https://en.wikipedia.org/wiki/Splay_tree.

[6] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Yuxi Cai, Fan Long, Andreas Park, and Andreas Veneris. Engineering economics in the conflux network. *arXiv preprint arXiv: 2004.13696*, 2020.

[8] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 523–540, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

[10] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[11] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[12] Aumasson JP. and Bernstein D.J. Siphash: A fast short-input prf. *Progress in Cryptology - INDOCRYPT*, 2012.

[13] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptology ePrint Archive*, 2015:1019, 2015.

[14] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, 2016.

[15] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[16] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for cryptocurrencies. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[17] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[18] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint*, 1805.03870, 2018.

[19] Chenxing Li, Fan Long, and Guang Yang. GHAST: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks. *arXiv preprint arXiv:2006.01072*, 2020.

[20] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[21] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[22] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[24] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 817–831, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Kartik Nayak, Srijan Kumar, Andrew Edmund Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *IEEE European Symposium on Security and Privacy*, pages 305–320, 2016.

[26] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

[27] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324. ACM, 2017.

[28] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[29] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. *CoRR*, abs/1507.06183, 2015.

[30] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: Serialization of proof-of-work events: confirming transactions via recursive elections, 2016.

[31] Yonatan Sompolinsky and Aviv Zohar. Phantom, a scalable blockdag protocol. https://eprint.iacr.org/2018/104.pdf.

[32] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[33] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, 2019. USENIX Association.

[34] WonderNetwork. Global ping statistics: Ping times between WonderNetwork servers. https://wondernetwork.com/pings, Apr. 2018.

[35] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07), 2017. Accessed: 2018-01-03.

[36] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. OHIE: Blockchain scaling made simple. *arXiv preprint arXiv:1811.12628*, 2018.

[37] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding.

Reconstructing proprietary video streaming algorithms

Maximilian Grüner, Melissa Licciardello, Ankit Singla
Department of Computer Science, ETH Zürich

Abstract

Even though algorithms for adaptively setting video quality in online streaming are a hot topic in networking academia, little is known about how popular online streaming platforms do such adaptation. This creates obvious hurdles for research on streaming algorithms and their interactions with other network traffic and control loops like that of transport and traffic throttling. To address this gap, we pursue an ambitious goal: reconstruction of unknown proprietary video streaming algorithms. Instead of opaque reconstruction through, *e.g.,* neural networks, we seek reconstructions that are easily understandable and open to inspection by domain experts. Such reconstruction, if successful, would also shed light on the risk of competitors copying painstakingly engineered algorithmic work simply by interacting with popular services.

Our reconstruction approach uses logs of player and network state and observed player actions across varied network traces and videos, to learn decision trees across streaming-specific engineered features. We find that of 10 popular streaming platforms, we can produce easy-to-understand, and high-accuracy reconstructions for 7 using concise trees with no more than 20 rules. We also discuss the utility of such interpretable reconstruction through several examples.

1 INTRODUCTION

Video streaming is one of the most popular Internet services today, forming a majority of downstream Internet traffic [35]. Naturally, there is great interest in optimizing video delivery. One particularly well-studied problem is that of designing adaptive bitrate algorithms that adjust video quality in response to changes in network bandwidth. ABR algorithms attempt to maximize some notion of quality-of-experience, which typically combines video playback metrics like average video quality, and playback interruptions and stability.

The problem of maximizing QoE in video streaming is crisply defined, intellectually interesting, and practically valuable. Thus, numerous ABR algorithms have been suggested in recent work to tackle it, *e.g.,* Oboe [3] and MPC [46]. However, little is known about the proprietary algorithms actually deployed in widely used video streaming services such as YouTube, TwitchTV and Netflix.[1] We attempt to address this gap by exploring whether it might be possible to learn such algorithms by controlled observation of video streams.

Our goal is to produce ABR controllers that: (a) mimic the observed behavior of ABR logic deployed in target online video services across a wide range of network conditions and videos; and (b) are open to easy manual inspection and understanding. Note that the latter precludes the direct use of blackbox machine learning techniques like neural networks.

We are motivated by three factors. First, this effort helps understand the risk of competitors copying painstakingly-engineered algorithmic work simply by interacting with popular, public-facing front-ends. Second, being able to reconstruct widely deployed algorithms would allow head-to-head comparisons between newly proposed research ABRs and industrial ABRs, something lacking in the literature thus far. Third, given that video is the majority of Internet traffic, this traffic being controlled by unknown proprietary algorithms implies that we do not understand the behavior of most Internet traffic. This makes it difficult to reason about how different services share the network, and interact with other control loops such as congestion control and traffic shaping.

The above use cases help sharpen the goals for our reconstruction effort. Simplifying our task is the fact that instead of exact algorithm recovery, we need functional equivalence of a reconstruction with its target algorithm over a large, varied set of inputs – note that the same set of outcomes could be arrived at by two substantially different algorithms, making exact recovery of a particular algorithm impossible. However, our use cases also impose a difficult additional requirement: our reconstructions must be human-interpretable, allowing not only the mimicking of observed behavior, but also manual inspection and understanding. A competitor seeking to copy the ABR logic of an online service needs interpretability to be able to modify it as necessary for their use.[2] They would also like to ensure the robustness of the obtained logic, something that is difficult with blackbox learning — prior work has shown corner cases with undesirable behavior in blackbox learning methods applied to networking [17]. Likewise, in terms of comparisons between industrial and academic ABRs, we would not only like to observe the performance

[1] Researchers at Netflix published, in 2014, work on this problem [15], including tests on their commercial deployment. Per our conversations with them, their current deployment incorporates some features of this published work, but they are unwilling to share more details, including the differences from this published approach.

[2] Our work enables an understanding of whether this risk *exists*: "Can a competitor reconstruct an ABR in a meaningfully beneficial, robust way?" We leave the question of how this risk may be tackled to followup work.

differences empirically, but also understand where they stem from. Lastly, reasoning about interactions with other network control loops and competing services also requires having a richer understanding of the control logic under study than blackbox learning can provide.

Algorithmic reconstruction of this type is an ambitious goal, with the current tools available for general-purpose program synthesis still being fairly limited. However, there are two reasons for optimism if we can suitably narrow our scope: (a) the core of ABR algorithms involves a small set of inputs, and has a limited decision space; and (b) it is easy to collect large amounts of curated data for analysis.

Our approach automatically generates concise, human-interpretable rule-sets that implement ABR by learning from an existing target ABR algorithm. These rule-sets map the client and network environment, video features, and state over the connection, to a video quality decision for the next video chunk. To obtain generalizable, succinct, and interpretable pseudocode in a reconstruction, we find that it is insufficient to directly use sophisticated techniques from imitation learning [5, 34]. As we shall show later, such methods can either mimic the behavior of a target accurately with a large set of complex rules, or, when limited to a small set of rules, lose accuracy. Our approach sidesteps this tradeoff by embedding suitable domain knowledge in the learning mechanism: framing intuitive primitives familiar to domain experts, and making them available to the learning mechanism, results in rule-sets that are accurate, concise, and meaningful.

We use our approach to obtain concise reconstructions that can successfully mimic the decision-making of several target academic and industry ABR algorithms, achieving high agreement with their decisions and similar video QoE behavior. Of the 10 online streaming services we evaluate across, our reconstruction achieves behavior similar to its target for 7 services. In each case, we produce a concise decision-tree with 20 or fewer short rules, using primitives that are intuitive and easy to understand. We also explain the reasons for failure for the remaining 3 services.

We make the following contributions:

- We describe an approach for deriving accurate *and* concise rule sets for ABR, using a corpus of decision outcomes over network traces and videos. Our approach handles the complex output space corresponding to diverse video encodings, as well as noise in the data.
- We apply our method to the reconstruction of algorithms deployed in 10 popular streaming services. For 7 services, we successfully achieve high agreement with their decisions and closely similar streaming behavior.
- The rule sets we obtain are concise, with 20 or fewer rules in each case. Our code also generates a loose natural language translation, which we used extensively in understanding problems and improving performance.
- We also expose a likely fundamental compromise necessary for interpretable and effective learning: the time-consuming encoding of domain knowledge.
- Our code and reconstructed ABRs are open-source [12].

Beyond the above results, our ambitious effort raises several exciting questions for future exploration, such as: (1) on the tradeoffs between the effort invested in embedding domain knowledge, and the quality of the inferred pseudocode; (2) to what extent such domain knowledge may itself be learnt from a corpus of hand-designed algorithms broadly from the networking domain; (3) applying our approach to other networking problems, like congestion control, and newer problems where we have more limited experience, such as multipath transport; (4) and how online service providers may obscure their logic against reconstruction, if so desired.

2 RELATED WORK

Numerous high-quality ABR proposals have appeared just within the past few years [3, 11, 31, 37, 45], but relatively little is known about widely deployed industrial ABR algorithms.

There is a large body of work on reconstructing unknown algorithms. One may approach this using code analysis, like Ayad et al.'s analysis of Javascript code for some online video services [16]. However, some targets can be too large and obfuscated for such analysis – YouTube, for instance, comprises 80,000+ lines of obfuscated Javascript. We used JS NICE [36], the state-of-the-art in Javascript deobfuscation, but even coupled with a step-through debugger and with help from the authors of JS NICE, this provided little insight – ultimately, manually examining such a large piece of code with meaningless variable names to reconstruct its functionality seems futile. It also has the downside of potentially requiring substantial rework for even small changes in the target. Even more fundamentally, the code may not be available at the client at all, with decision-making residing on the server side.

Several prior efforts have used manual experimentation and analysis for dissecting the behavior of a variety of online services [2, 9, 16, 19, 21, 27, 44]. For instance, Mondal et al. [27] used network traces to experimentally study the behavior under changing network conditions, and then manually draw coarse inferences, such as that YouTube's requested segment length varies with network conditions. An earlier effort on inferring Skype's adjustment of its sending rate [19], was based on the researchers making experimental observations, then manually hypothesizing a control law, and finally tuning its parameters to fit the data. Our own parallel measurement study [21] experimentally examined the behavior of several deployed ABR algorithms in terms of metrics like stability of playback and convergence time after bandwidth changes. In concurrent work, Xu et al. [43] propose a method for inferring the quality of video chunks downloaded within encrypted streams, and apply it to experimentally study the streaming outcomes in response to different traffic throttling schemes. In contrast to all the above efforts, our goal here is to *automatically generate logic that mirrors a target ABR algorithm's behavior* by observing the target ABR's actions in response to variations in the environment and inputs.

There are also efforts in networking to inspect the internals of learning-based networked systems. This work is not directly applicable to our goal of reconstructing arbitrary ABRs, which are most likely non-ML, and more importantly, are not available to us. However, one could first train a blackbox-ML algorithm to mimic any reconstruction target, and then use such tools. Recent work on inspecting [10] or verifying [17] systems built using ML has examined Pensieve [23]. The authors frame hypotheses/questions about the system's behavior, and then evaluate them. However, this (a) requires knowing what hypotheses to examine, and (b) does not yield a reconstruction. Among efforts in this vein, the most closely related are the concurrent TranSys [26] and PiTree [25] studies. PiTree focuses on converting ABR algorithms to decision trees, and TranSys broadens this approach to NN-based strategies in networking. Both are networking applications of a broader paradigm in ML, which we discuss next.

Beyond networking efforts, *imitation learning* is a rich discipline in its own right. Most work in this direction uses (uninterpretable) neural networks [6, 14, 42], but recent work has also developed model-free approaches to approximate the learned neural network via, *e.g.,* a decision tree [5, 34]. As we show later in §6.2, directly using this approach (like TranSys and PiTree) does not meet both of our accuracy and interpretability goals simultaneously, instead requiring the sacrifice of one or the other. While complex decision trees, with a large number of rules with many literals, can robustly imitate a target algorithm, they are difficult, if not impossible, for even domain experts to understand and work with. On the other hand, restricting the complexity of the generated trees results in a loss of imitation accuracy and robustness. While the expressiveness and compactness of these approaches can be improved by employing genetic algorithms to craft features for use therein [13], this often leads to both overfitting, and complex, non-intuitive features.

Lastly, program synthesis is a rich and growing field. While we use one particular strategy for ABR reconstruction, there are other tools we plan to examine in future work. The most promising perhaps is recent work combining learning with code templates [41], where the core idea is to modify templates to minimize the distance from a target learning algorithm. An alternative "deductive synthesis" approach, as employed in Refazer [33], could also be fruitful.

To the best of our knowledge, our work is the first to attempt an interpretable reconstruction of unknown deployed ABRs.

3 DATA PREPARATION

We extend a trace collection harness that we built for a measurement study, where we used manual analysis to comment on the behavior of deployed ABR algorithms across 10 streaming platforms [21].

We launch a video stream on a target service, and according to an input network trace, shape the throughput at the client using Linux `tc`. We record the current client buffer occupancy, the video chunk qualities played out, video metadata, etc. The client buffer occupancy is directly measured through the instrumentation of the HTML5 player element. If the HTML5 player element were not available, we could instead use the captured HTTP chunk requests (locally at the client, making encryption irrelevant) to reconstruct the buffer occupancy — this strategy may be of use for future work exploring mobile ABR implementations. This alternative can be less accurate though, as "redownloading" (*e.g.,* to replace already downloaded low-quality chunks in the client player buffer by higher-quality ones) introduces an ambiguity into which chunk is actually played.

For each platform, by appropriate tailoring of HTTP requests, we also fetch all chunks for the test videos at all qualities, such that we can use these videos in an offline simulation, allowing the learned ABR to make choices different from those in our logs, as well as to enable us to study the behavior of academic ABRs. Ultimately, we obtain the following measurements:

- $\mathbb{C}_t$: segment size (Mb) downloaded for request t
- $\mathbb{R}_t$: segment bitrate (Mbps) for request t
- $\mathbb{V}_t$: segment VMAF[3] for request t
- $\mathbb{D}_t$: download time for request t
- $\mathbb{Q}_t$: quality level requested in request t
- $\mathbb{S}_t$: segment length (seconds) downloaded for request t
- $\mathbb{P}_t$: Percent of the video played at request t
- $\mathbb{B}_t$: buffer size (seconds) when requesting t
- $\mathbb{RB}_t$: rebuffer time (seconds) when requesting t
- $\mathbb{C}^i_{t+n}$: segment size of quality i for n^{th} chunk after t
- $\mathbb{R}^i_{t+n}$: segment bitrate of quality i for n^{th} chunk after t
- $\mathbb{V}^i_{t+n}$: segment VMAF of quality i for n^{th} chunk after t

4 RULE-SET BASED INFERENCE

We shall first consider a motivating example for why rule-sets are a simple and potent representation for our type of target algorithms, and then present our recipe for constructing succinct rule-sets that capture the target algorithm's behavior.

4.1 Motivating example

Let us examine a simple throughput-based ABR algorithm, similar to that described in prior work [15]. It uses only the throughput estimate for the last video chunk fetched, $\mathbb{T}_{N-1}$,

[3] VMAF is a video perceptual quality metric [20].

$$\begin{aligned}
\text{quality } 0 &\rightarrow \mathbb{T}_{N-1} \leq 4.99 \\
\text{quality } 1 &\rightarrow \mathbb{T}_{N-1} > 4.99 \;\&\; \mathbb{T}_{N-1} \leq 6.97 \\
\text{quality } 2 &\rightarrow \mathbb{T}_{N-1} > 6.97
\end{aligned}$$

Fig. 1: Minimal rule-set for a reservoir-based algorithm, which uses only the last chunk's throughput estimate to pick a quality level.

and two thresholds: *reservoir* and *cushion*. If $\mathbb{T}_{N-1}$ < *reservoir*, the lowest quality is served. If $\mathbb{T}_{N-1}$ > *reservoir* + *cushion*, the highest quality is served. For other values of $\mathbb{T}_{N-1}$, quality is linearly interpolated between these levels.

This algorithm, regardless of its specific instantiation with particular values of the thresholds, can be easily expressed as a set of rules. For a simple instantiation with only 3 quality levels, and both *reservoir* and *cushion* set to 4 Mbps, this rule-set is shown in Fig. 1.[4] The rule-set is inferred (which is why the rules contain imprecise values like 6.97) by the process we shall describe shortly.

We caution the reader against concluding from this small motivating example that only simple, stateless, "templates with parameters / thresholds" type of algorithms can be expressed in this way. Rule sets are capable of capturing complex stateful behavior, as long as primitives encoding this state are made available for their use.

4.2 Decision trees and rules

We first learn binary decision trees [18] that encode conditions for specific outputs, *e.g.,* the video quality levels requested. Further, in such a decision tree, each path from the root to a leaf can be naturally interpreted as a descriptive rule capturing the conditions for the outcome at the leaf to be reached.

Consider a single-threshold decision: "If throughput < 5 Mbps, pick low quality; otherwise, pick high quality.". This can be captured in a 3-node tree with the conditional at its root, and the two outcomes as leaves. In this case, the rule-lengths, *i.e.,* the path lengths from the root to the leaves, are 1; and so is the number of "splits" in the tree.

Fig. 2 shows a more complex target decision plane with two inputs (along the x and y dimensions), where there are still only two outcomes (labels), but the data samples that map to these labels are separated by more complex logic. If we constrain the decision tree that approximates this decision plane to use rules of only length one, we can use only one line separating the labels, as shown in the top-left smaller figure. Allowing more and more complex (longer) rules, allows a tighter representation of the target decision plane. Of course, using too many rules risks overfitting, especially under noisy data that is typical in networking. Fortunately, our goal to obtain concise rule sets aligns well with that of avoiding overfitting and preserving generalization.

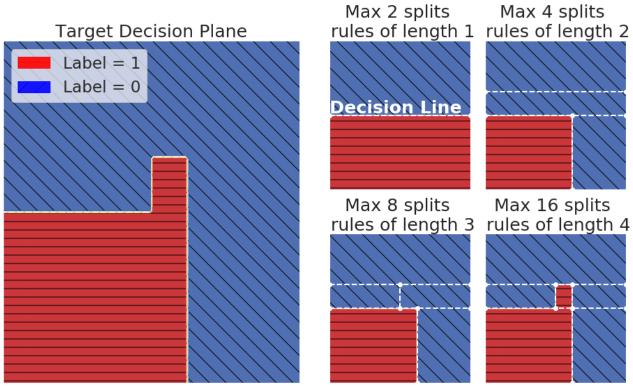


Fig. 2: The big image is the target decision plane, with 2 labels. On the right are its approximations with decision trees of different rule-lengths, going from 1 to 4.

Framing the output space: A key design consideration in using decision trees is the framing of the output decision space. Suppose we frame decision outcomes in terms of the video quality level that the client should fetch the next video chunk at. *If* all the videos being served were encoded with the same quality levels, both in terms of number and their bitrates, *e.g.,* 6 video qualities at bitrates of {200, 450, 750, 1200, 2350, 4300} Kbps, this would be easy: there are 6 *a priori* known outcomes that we can train decision trees for.

However, this is clearly overly restrictive: in practice, ABR algorithms must tackle a variety of videos encoded at different bitrates. The set of different bitrates at which a video is encoded in is referred to as its "bitrate ladder". Providers like Netflix even use per-video customization of bitrate ladders, varying the number and separation of bitrate levels [1]. This diversity in the output space is a challenge for learning approaches: what should we present as the potential output decision space? It is noteworthy that Pensieve [23] does not fully tackle this challenge, instead restricting the video bitrate levels to a small pre-defined set.

To overcome this issue, we formulate the decision process in terms of video quality being upgraded or downgraded relative to the current video quality. With one decision, a maximum of n quality shifts are permitted in either direction, with n being a tunable parameter. Of course, this prevents us from capturing some algorithms, where larger quality changes (than n) may be enforced in a single step. However, this is atypical, as video streaming typically targets smooth output. Even if some algorithm involves such decision making, our approach only differs from such a target's decisions transiently. This small compromise allows us to generalize to arbitrarily diverse video bitrate ladders.

[4] Readers may expect the rule-set in Fig. 1 to mean that *reservoir* $= 5$ and *cushion* $= 2$. The discrepancy stems from the discreteness of the interpolation: for some $\mathbb{T}_{N-1} >$ *reservoir i.e.,* $\mathbb{T}_{N-1} \in [4,5]$, quality 0 will be chosen.

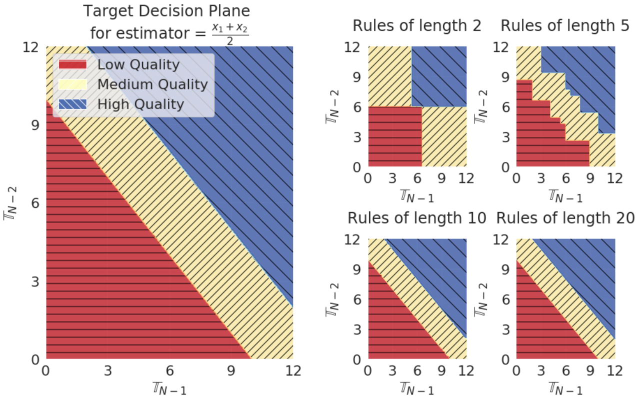


Fig. 3: Here, the target decision plane (big, left) is governed by the mean of $\mathbb{T}_{N-1}$ and $\mathbb{T}_{N-2}$. The smaller figures show that we need long rules to approximate this if we are restricted to using individual literals ($\mathbb{T}_{N-1}$ and $\mathbb{T}_{N-2}$) in our rules.

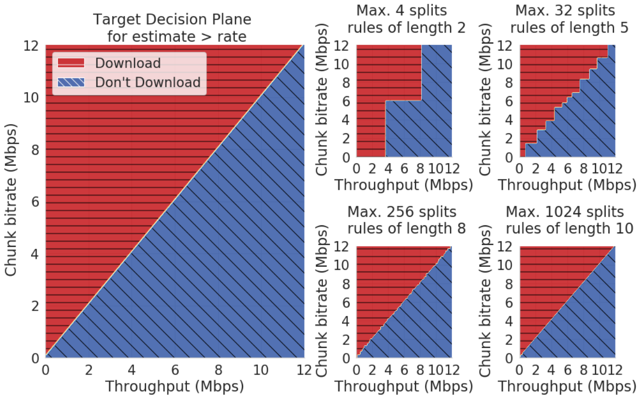


Fig. 4: A decision plane comparing the throughput estimate to the chunk bitrate is difficult to capture without long rules if rules can only be framed in terms of the individual literals.

4.3 Feature engineering

Applying textbook decision-tree inference, with the above framing, one can already infer simple algorithms. However, as we shall see, appropriate customization based on domain expertise is crucial to obtain concise and generalizable rules.

Consider, for instance, a target algorithm that uses the mean throughput across the last 2 video chunks fetched. Naively learnt rules will then contain complex conditionals across both $\mathbb{T}_{N-1}$ and $\mathbb{T}_{N-2}$. Fig. 3 shows this for rules of increasing length, up to 20. The target decision plane uses the mean, $\frac{\mathbb{T}_{N-1}+\mathbb{T}_{N-2}}{2}$ to decide between three video qualities. Rules of length 2 and 5 yield poor accuracy, necessitating much longer (complex) rules.

Of course, if we knew that the building block for throughput estimation is the mean, we could simplify such rules substantially by expressing them in terms of the mean. Thus, we can consider adding common estimators, based on our domain knowledge, to the feature-set available to our learning pipeline. Then, instead of only learning across primitive literals (like each individual chunk's throughput), more compact representation across these non-primitive literals becomes possible. We thus explore three classes of such non-primitive features that are intuitive and likely commonplace in ABR, and even more broadly, other networking algorithms.

Throughput estimators: Clearly, having the most accurate estimate of network throughput is advantageous in deciding on video quality. As such, throughput estimators are potentially useful building blocks. We consider two types of estimators. The first type is only parametrized by the number of past throughput measurements it aggregates using a mean, median, harmonic mean, etc., while the second type involves additional tunable parameters, such as the weight decrease, α, in an exponential weighted moving average (EWMA), which sets the relative weight of a new measurement compared to old measurements.

Encoding these estimators with a range of different parameter choices gives us a large set of features ranging from nearly stateless (*e.g.,* using only the last chunk's throughput) to those with long-term state (*e.g.,* a moving average). In addition to throughput, we also construct features capturing the *variation* across recent throughput measurements as it characterizes the stability of the network.

Comparisons: Decisions often depend on not just thresholding of certain primitive or non-primitive features, but also on comparisons among features. For instance, generalizing even a simple rate-based algorithm to work for arbitrary videos encoded with different bitrate ladders requires a comparison: is the throughput larger than a particular step in the bitrate ladder? Unfortunately, while decision trees can capture such comparisons using only primitive features, they require a large number of rules to do so. This is shown in Fig. 4, where the decision trees with rules of length 2 and 5 do not accurately represent a simple comparison-based target decision plane.

Thus, we must encode potential comparisons of this type as non-primitive features. These can also be parameterized in a similar manner to the throughput estimators discussed above, *e.g.,* by what *factor* should one feature exceed another.

Planning ahead: ABR, like many other control tasks, is not only about making one decision in isolation, but also considering its longer-term influence. For instance, it might be interesting to estimate the max, min, or mean rebuffering one should expect given the decision to download a chunk at a certain quality, assuming the throughput stays the same. We design features along these lines, such as QoE in MPC [46].

More features? Over time more features can be added to enhance our approach without having to reason about their mutual interactions, as would be the case with incorporating new ideas into human-engineered routines. One could also extend this approach by adding automatically engineered features [13]. However, maintaining interpretability would require limiting the complexity of auto-generated features.

5 IMPLEMENTATION

We implement the rule inference pipeline in Python3. For the decision tree, we use the standard implementation provided by the scikit-learn library [30]. If not otherwise mentioned we use a maximum of 20 rules and limit one-step changes in quality to upgrading or downgrading by at most 2 quality levels. The 20-rule limit is somewhat arbitrarily chosen as a quantitative threshold for interpretability, but we also find that for our approach, more rules do not improve performance substantively in *most* cases. This threshold is essentially a hyperparameter that could be tuned by practitioners based on where they seek to operate in the tradeoff space involving interpretability, avoiding overfitting, and performance.

Baselines: To put our results in context, we compare them against three neural network approaches, both as-is (blackbox approaches, always represented by a recursive neural network with GRU cells [7]), and translated to decision trees. The first blackbox approach is the simplest, attempting to directly copy the decisions in a supervised learning setting. The other two use more sophisticated imitation learning methods [14, 42].

For translating the blackbox approaches into decision trees, we test two state-of-the-art methods [5, 34]. One of these [5] needs a reward function. In the vein of other imitation learning approaches, we use a clustering algorithm to assign a similarity reward to every sample. In our implementation we use an isolation forest [22] implemented in scikit-learn [30] with the standard parameters as our clustering approach. At every training step, we sample 3000 samples (as this gave the best results) according to the cluster weighting. We also tried changing the weighting function to a more agnostic divergence measure as the proposed decision by the blackbox approach might not always be what the original algorithm had in mind. This makes the sampling approach more robust.

For each reconstruction, when we compare results to our approach, we use the best blackbox approach and the best tree-translation. Thus, we invested substantial effort in implementing sophisticated baselines from the ML literature.

We also test whether our approach benefits from learning from the blackbox, instead of directly from the data. We find that this yields only minor improvements for 2 of our 10 reconstruction targets. We also explore learning in two passes, where in the first pass, we learn a tree over engineered features, and use a classifier to label its decisions in terms of their similarity to decisions made by the reconstruction target. In the second pass, we re-learn across weighted data samples, such that samples corresponding to more similar decisions are weighted higher. This approach also results in only minor improvements for one of our ten reconstruction targets.

Feature engineering: We instantiate our features (§4.3) with appropriate parameter ranges as below. 'Action' refers to quality decisions, such as maintaining quality, or increasing or decreasing it by up to n quality levels. The 'any' operator instantiates all options for a parameter or primitive.

1. Standard deviation, mean, harmonic mean, EWMA, and q^{th} percentile over the last $\boldsymbol{n}$ chunks, with $n \in \{1 \ldots 10\}$. Additionally, for EWMA, $\boldsymbol{\alpha} \in \{0.15, 0.35, 0.55, 0.75, 0.95\}$.
2. For q^{th} percentile, $\boldsymbol{q} \in \{15, 35, 55, 75, 95\}$.
3. Reward $\mathcal{R}$ achievable by planning ahead 3 steps for **any** action with **any** throughput estimate. The 'any' operators here imply that we have numerous reward features, each of which combines one of the many available throughput estimators (from 1. and 2. above) with one of the possible actions. As the reward function, we use the linear QoE function introduced by Yin et al. [46], which converts bitrate, buffering and bitrate change per chunk downloaded into a score. Note that this is not necessarily what any of our reconstruction targets is optimizing for – each provider may have their own reward goals. We use this feature simply as a compact representation of QoE components.
4. Fetch time for **any** action, **any** throughput estimate.
5. Bitrate gained weighted by the buffer filling ratio for **any** action, **any** throughput estimate. Intuitively, this captures the gain in bitrate relative to its cost, *i.e.,* how much the buffer is drained by an action if throughput stays the same.
6. VMAF gained weighted by the buffer filling ratio for **any** action, **any** throughput estimate. Same as above, but with VMAF.

Ultimately we make ~ 1300 features available to the learner. Note the multiplicative effect of the **any** operator above.

Throughout, we use a standard training, validation, and testing methodology. The test set contains two videos combined at random with 60 traces randomly sampled from the overall set; these 60 traces are *neither* in the training nor in the validation set. We only discuss results over the test set.

Automated Feature Engineering: As a comparison and future outlook on the possibility of automated feature engineering, which has shown promise in other applications [13], we also coarsely implement an automated feature generator. This generator recombines the raw features in an iterative fashion so that the most used features "survive" and get recombined and the others "die" out. We use the library gplearn [39] with basic mathematical operators as usable functions. We limit the iterations to $s \in [50, 100, 150]$ seconds to avoid overfitting.

6 EVALUATION

We summarize below the experiments we conducted as well as their top-line conclusions:

1. How well can we reconstruct target algorithms? We can mimic the decision-making of 7 of 10 targets to a high degree, and obtain high similarity scores.
2. What influence does domain knowledge have? Certain engineered features are crucial to obtain rules that gen-

eralize beyond training data, are concise, and achieve similar QoE as the target algorithms.

3. How interpretable and generalizable are the output rule sets? We find that we can easily spot flaws in the learned algorithm and propose ways to adapt it. Further, trees with only 20 leaves suffice in most cases.
4. How do deployed ABRs compare to academic ones? We find that academic ABRs generally outperform industrial ones, with the caveat that our evaluation uses metrics from academic work. Interestingly, we observe that one provider's algorithm shows behavior closely matching the well known BOLA algorithm, indicating potentially that this provider uses BOLA or a derivative of it.

6.1 Experimental methodology

Target platforms: We use the same 10 popular streaming platforms we used in our measurement study of deployed ABRs [21]. While certainly not exhaustive, this is a diverse set of platforms, including some of the largest, such as Twitch and YouTube; some regionally focused, such as Arte, SRF, and ZDF; and some serving specific content verticals, such as Vimeo (artistic content), TubiTV (movies and TV), and Pornhub and XVideos. We exclude Netflix, Hulu, and Amazon Prime because their terms of service prohibit robotic interaction with their services [21].

Different platforms encode content at varied resolutions and number of resolutions, ranging from 3 quality levels for TubiTV to 6.5 on YouTube (on average across our test videos; YouTube has available resolutions for different videos.) For Twitch, which offers both live streams and video-on-demand of archived live streams, we only study the latter, as live streaming is a substantially different problem, and a poor fit with the rest of our chosen platforms. For several of the providers we study, there are multiple implementations, such as for desktop browsers, mobile browsers, or mobile apps; we only attempt reconstruction for the desktop versions.

We also evaluate our ability to emulate well-known academic approaches for ABR. We use the Robust-MPC (henceforth, just MPC throughout) and Multi-Video Pensieve (henceforth, NN, because it uses a neural network approach) implementation provided by the authors of the Pensieve paper [24]. We train and test these approaches on the Twitch video data set. To speed up our experiments, for MPC, we use a lookahead of 3 chunks instead of 5, finding that this did not make a large difference in performance.

Videos: The type of content can have a substantial bearing on streaming performance, *e.g.,* videos with highly variable encoding can be challenging for ABR. We thus used a set of 10 videos on each platform. Where a popularity measure was available, we used the most popular videos; otherwise, we handpicked a sample of different types of videos. Videos from each platform are encoded in broadly similar bitrate ranges, with most differences lying at higher qualities, *e.g.,* some content being available in 4K.

Network traces: Our experiments use synthetic and real-world traces from 3 datasets in past work [3, 8, 32]. Unfortunately, a full cross-product of platform-video-trace would be prohibitively expensive — the FCC traces [8] alone would require 4 years of streaming time. To sidestep this while still testing a diversity of traces, we rank traces by their throughput variability, and pick traces with the highest and lowest variability, together with some randomly sampled ones.

Our final network trace collection consists of the 5 least stable, 5 most stable, and 20 random traces from the Belgium trace collection [40]; and 10 most/least stable ones plus 25 random traces from each of the Norway [32], the Oboe [3] and the FCC datasets.[5] We also use 15 constant bandwidth traces covering the range from 0.3 to 15 Mbps uniformly. Lastly we add 10 step traces: after 60 seconds of streaming we suddenly increase/drop the bandwidth from/to 1 Mbps to/from 5 values covering the space from 1.5 to 10 Mbps uniformly. If a trace does not cover the whole experiment, we loop over it.

In total, we use 190 traces with throughput (average over time for each trace) ranging from 0.09 to 41.43 Mbps, with an average of 6.13 Mbps across traces. Note that we make no claim of our set of traces being representative; rather our goal is to test a *variety* of traces.

Evaluation metrics: For training our approach and evaluating its accuracy in a manner standard in learning literature, we use two metrics: one measures *agreement*, and another the *similarity* of sets of decisions. We train towards maximizing the harmonic mean of these. Additionally, for our ABR-specific use-case, we evaluate the video quality of experience [28].

Agreement, F_1 score: For each output decision, we compute the precision and recall of the inferred algorithm against its target. The F_1 score is the harmonic mean of these. $F_1 \in [0,1]$, with 1.0 being optimal. We compute an average over the F_1 scores across the labels in an unweighted fashion.

What is high/low agreement? If we were not interested in interpretability, we could obtain a procedure that mimics any target algorithm by using blackbox learning. We can think of the agreement such a blackbox approach achieves with its target as a baseline free of our conciseness constraint. On the other end of the spectrum, if the inferred rules do not achieve substantially higher agreement with the target than a generic 'reasonable' algorithm, then they are useless: this implies any reasonable algorithm would make at least that many decisions similar to the target. We use the simple rate-based approach as the concrete stand-in for this generic reasonable algorithm.

Similarity: As we cannot assume anything about how each provider designs their algorithm, we must use an agnostic approach in evaluating whether the experience under our reconstruction and the actual ABR is the same. Thus, we choose, as is typical in imitation learning, to learn whether the ex-

[5] Specifically, the stable collection from September 2017 [8].

perience of two streaming sessions is "similar". Similarity measures whether a set of samples (our reconstruction's decisions) is likely to be from a given distribution (the actual ABR's decisions). To classify whether a particular decision looks like it has been taken by the actual ABR or by our reconstruction, we choose an isolation forest [22].

Each of these two metrics is insufficient on its own. High agreement is useful, but making a few important decisions differently can substantially change a video stream's behavior. Thus the need for similarity. However, there's a benign solution to achieving high similarity: most commercial providers tend to keep the quality stable, so, by just keeping the same quality one can get high similarity. Conversely, agreement solves this problem: to get high agreement, we must match a large fraction of *each* decision type, matching only the "keep quality" decisions will result in poor agreement because of low matches on quality changes.

QoE: Agreement and similarity can be thought of as "micro-benchmarks" – these are standard measures in imitation learning, and are useful both for training our approach, and evaluating its learning accuracy. But ultimately, we also want to know: "How different is the user experience from an ABR versus from our reconstruction of it?". We thus directly compare how well different components of a visual QoE metric used in earlier work [28] match up between a target and its reconstruction. As we show below, agreement and similarity correlate well with QoE: when a reconstruction achieves high agreement and similarity, it typically also performs like the target algorithm in terms of different QoE components.

Finally, we also comment on the role of domain knowledge in achieving good results with our approach, and the interpretability of our reconstructions.

6.2 Results

Agreement and similarity, Fig. 5: We compare the agreement and similarity achieved by our rule-set approach against the (best) blackbox approach and the simple rate baselines across all 10 online providers. We also include MPC and Pensieve (NN) evaluated on the Twitch videos.

The rule-sets achieve nearly the same or better agreement than the blackbox approach achieves for a reconstruction target in each case – in the worst case (NN), the rule-set's agreement score is 8% lower. Note that in many cases, we achieve higher agreement than even the blackbox approach. This is due to the imitation learning approaches trying to achieve higher similarity in terms of behavior rather than matching each individual quality decision.

The rule-sets also achieve high similarity in most cases, in the worst case (Twitch), achieving a ≈20% lower similarity score than the best blackbox approach, and in the mean, 5% higher. In contrast, the rate-based approach achieves not only very low agreement, but also very poor similarity.

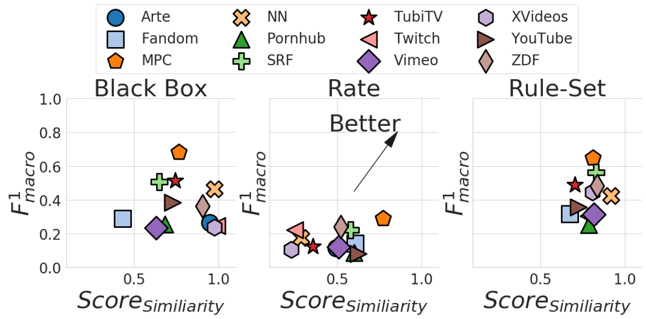


Fig. 5: The generated rule-sets are never worse by more than 8% and 20% than the blackbox approach on agreement and similarity respectively. In contrast, the rate-based approach achieves extremely poor results.

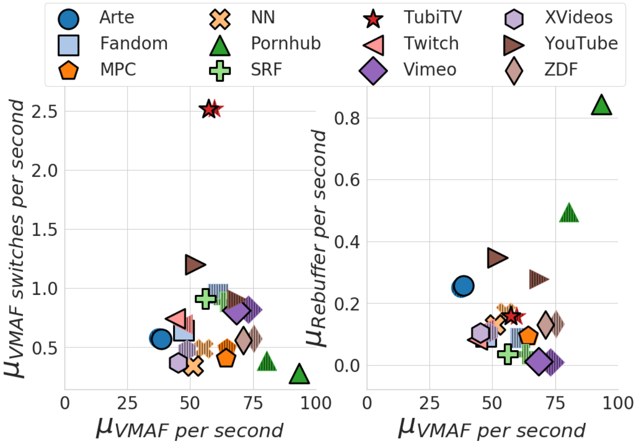


Fig. 6: For all but 3 of the 12 targets, the reconstruction matches the target algorithm very closely. For YouTube, Fandom, and PornHub, there is a substantial difference in performance; these are the same 3 providers in the bottom-left of Fig. 5, for which we achieve the lowest agreement and similarity scores as well.

Some readers may interpret the "low" absolute numbers in Fig. 5, *e.g.,* $F_1 \sim 50\%$, as a negative result. However, note that F_1 differences often don't cause appreciable video session quality differences, *e.g.,* if an ABR switches two quality levels in one step, and its reconstruction switches them in two successive steps, the F_1 score is lowered *twice*, but the video stream behavior changes negligibly. Also, rare labels (*e.g.,* increase quality by three levels) contribute equally to F_1 as common ones (*e.g.,* retain quality), so a few errors on rare labels have out-sized effect.

Video session quality metrics, Fig. 6: We compare metrics used to quantify video playback quality — VMAF [20], VMAF switches, and rebuffers – as seen in the actual algorithm (hatched points in the plot) and its rule-set reconstruction (solid points) across the same set of ABRs as in Fig. 5. For 9 of 12 targets, we achieve a very good match: the mean VMAF (x-axis in Fig. 6) for these 9 reconstructions is within 6% of the target ABR's on average; the maximum VMAF difference is 12%. These good matches include Twitch, SRF, Arte, ZDF, TubiTV, XVideos, Vimeo, MPC, and Pensieve (NN). On the other hand, for the other 3, YouTube, PornHub, and Fandom, there are large discrepancies, with quality metrics being very different for the reconstruction compared to the target. That our reconstruction does not yield good results on these targets is also supported by exactly these ABRs being in the low-agreement-low-similarity part of Fig. 5 (bottom-left in the rightmost plot). We further investigated these 3 negative results:

1. YouTube, in addition to making quality decisions, varies its segment length and can also redownload low-quality chunks to replace them with high-quality ones [27]. Ultimately, learning approaches will not frame new decision spaces, only logic for arriving at the asked-for decisions – in essence, YouTube is solving a different problem than we expected. This is a fundamental gap for efforts like ours: if the *decision space* is not appropriately encoded, outcomes will be sub-optimal. We could add the relevant primitives to achieve better results, but we resist such modifications that use the benefit of hindsight.
2. In a similar vein, we find that PornHub often switches to a progressive download, disabling video quality adaptation altogether. Our approach ends up overfitting to the progressive behaviour as we see few switches. If we exclude data where adaptation is disabled, we're able to match PornHub to within 4%, 0%, and 5% difference in terms of mean VMAF, rebuffering, and switching respectively.
3. For Fandom, we find that the issue is the limited complexity of our tree. A rule-set with a somewhat higher complexity (31 rules) performs substantially better, diverging from the target algorithm by 5%, 11%, and 22% in terms of mean VMAF, rebuffering, and switching respectively. Note that rebuffering and switching, being infrequent events are extremely difficult to *always* match, so a somewhat larger difference there is expected. As noted earlier, the rule-count is a hyperparameter that may need tuning for certain providers.

Role of domain knowledge, Fig. 8, 7: We already discussed in §4 why the use of domain knowledge is critical for interpretation: without simple primitives like moving averages, rules are framed in terms of various basic literals, resulting in complex and incomprehensible rules. Besides interpretation, we find that there is also substantial impact on agreement from adding useful domain knowledge.

We used our modified version of the DASH player to evaluate how the different trees emulating robust MPC generalize to other videos. We selected a mixed subset of 60 traces, that both include challenging and stable throughput measure and generated the distribution across them of linear QoE used in the MPC work [28]. Results are normalized to the mean QoE for the ground-truth MPC implementation across the same video-trace set.

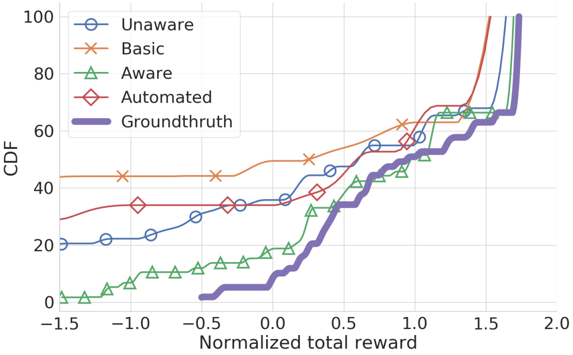


Fig. 7: Domain knowledge helps the rule-set (Bitrate-QoE).

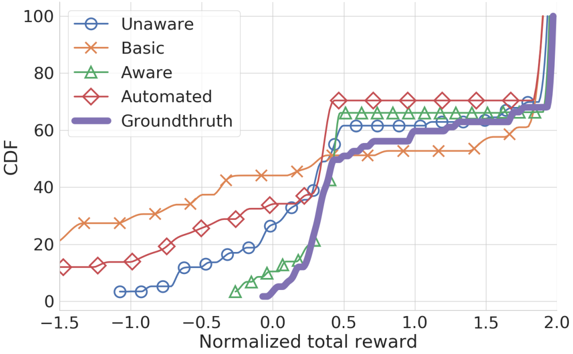


Fig. 8: Domain knowledge helps the rule-set (VMAF-QoE).

Fig. 7 shows how the rule-set reacts to additional building blocks being available for reconstruction in the form of engineered features. The 'Basic' rule-set is framed directly on the data features listed in §3, without any feature engineering. The 'Unaware' and 'Aware' approaches use several engineered features, as described in §4. The difference between them stems from the 'Unaware' approach only using engineered features related to buffer filling and draining in addition to the primitive data features. The 'Aware' approach with the benefit of all engineered features matches MPC the closest. 'Aware' improves average QoE over 'Unaware' by ∼5×. Thus, encoding domain knowledge helps not only with conciseness, but also performance and generalization. Also of note is the 'Automated' approach, which starts with the 'Basic' features, but can recombine them in the fashion described in §5. While promising for future exploration, it presently performs worse than manually engineering features, and does not produce features that are meaningful to humans.

Fig. 8 repeats the above experiment, but for a VMAF-based QoE function. The results are similar to those for bitrate -QoE. The average QoE of the 'Aware' reconstruction is within 10% of that of the target MPC algorithm, the median being within 2%. This is especially significant because we did not engineer any explicit features similar to this QoE function.

Interpretability: Across our efforts on reconstruction, the

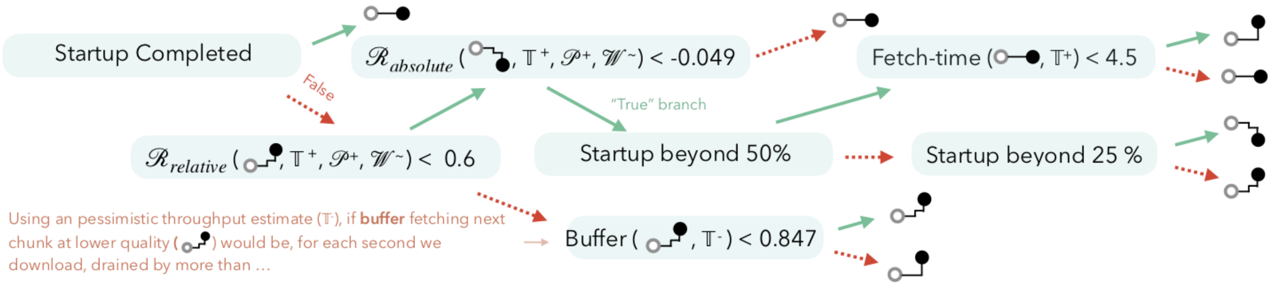

Startup Completed
$\mathscr{R}_{absolute}$ (, $\mathbb{T}^+$, $\mathscr{P}^+$, $\mathscr{W}^-$) < -0.049
Fetch-time (, $\mathbb{T}^+$) < 4.5
False
"True" branch
$\mathscr{R}_{relative}$ (, $\mathbb{T}^+$, $\mathscr{P}^+$, $\mathscr{W}^-$) < 0.6
Startup beyond 50%
Startup beyond 25 %
Using an pessimistic throughput estimate ($\mathbb{T}^-$), if buffer fetching next chunk at lower quality ( ) would be, for each second we download, drained by more than …
Buffer (, $\mathbb{T}^-$) < 0.847

Fig. 9: The core of the decision tree generated by learning from the SRF example data. The green (solid) and red (dashed) arrows are the "True" and "False" evaluations of the predicates at the nodes. The video quality of the last fetched chunk is denoted by ○ and the next chunk's by ● . Potential decisions and decision outcomes are coded in terms of the relationships between these qualities: *e.g.,* ○↗● denotes that the next chunk is requested at one higher video quality level. The predicates are in terms of expected buffer size after a certain potential decision, based on throughput estimates (*e.g.,* $\mathbb{T}^+$ is an aggressive/optimistic estimator); or on a reward ($\mathscr{R}_{relative}$) calculated relative to the other obtainable rewards involving throughput, rebuffering penalty ($\mathscr{P}$), the lookahead horizon over which these are estimated ($\mathscr{W}$), etc.

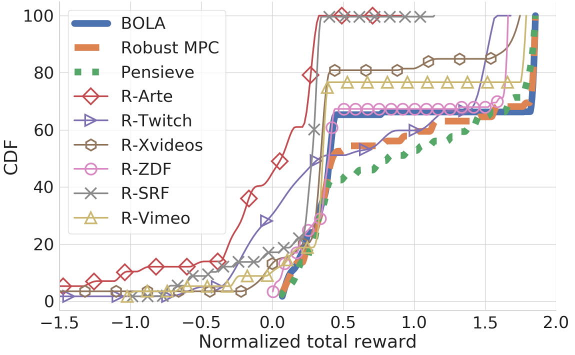


Fig. 10: Reconstructing commercial ABR algorithms allows us to uniformly compare them to both other commercial and academic ones under the same test conditions.

generated rule sets are concise, with no more than 20 rules. We realized early that being able to read and understand the generated trees would make debugging and improvements easier, and thus wrote a small, simple utility to translate the predicates in trees loosely into natural language. Fig. 9 shows an illustration of a tree generated for SRF. This version is hand-drawn for aesthetic reasons, but there is no fundamental reason it could not be auto-generated. Due to space constraints, this version is a compressed tree which was allowed to have at most 10 leaves instead of 20. We extensively examined and used our natural-language trees ourselves throughout this work, as we describe in a few instances in §7.

We also understand, to some extent, *why* small rule-sets suffice: (a) a single rule has implications capturing more than is plainly visible, *e.g.,* if the buffer is running too low, the best recourse is to lower quality, and not much new will be learnt from a long planning horizon; and (b) the domain-specific primitives are a dense encoding of useful knowledge. We caution readers against generalizing these observations to imply that small rule-sets will necessarily suffice for other problems where learning is effective — our exploration and results are limited to ABR. That small rule-sets would suffice for many ABRs, is also supported by prior work [10] showing, for instance, that the neural network ABR approach, Pensieve, largely depends on only 3 features.

Comparing academic and industry ABRs, Fig. 10: For targets we can reconstruct well, having a reconstruction enables us to compare them to each other and academic ABRs in the same test harness. This is non-trivial without a reconstruction, as each video platform has a very different streaming setup in terms of encoding, server architecture, and player design. For instance, if one platform uses more encoding levels than another, then the same ABR algorithm can make more fine-grained decisions on this platform than on the one with coarser encoding. Therefore the same algorithm on the same video would perform differently across video platforms, making it difficult to compare ABRs across providers without removing such confounding factors in a common test harness.

To this end, we extend the DASH architecture [37] with implementations of the (rule-set) reconstructions for the 6 targets we are able to match most closely. The same setup has ground truth implementations for BOLA [38], MPC [46], and Pensieve [23]. We evaluate VMAF-QoE [28] using the Envivio video typically used in academic work, and normalize the results to the mean QoE for Pensieve.

As the results in Fig. 10 show, Pensieve and MPC generally outperform the deployed ABRs' reconstructions, although, for a subset of traces, R-Twitch achieves the same or better performance as MPC. This is perhaps not unsurprising: we are evaluating all providers with QoE functions used in academic literature, while the providers may, in fact, be optimizing for a different goal. Amongst the providers, R-Arte's ABR achieves the worst performance on this QoE metric.

But perhaps most striking is the distribution-wide extremely close match between R-ZDF and BOLA – except for a few outliers at the tails, for most video-trace pairs, their performance is virtually identical. Thus, it is likely that ZDF is using BOLA, or a derivative of it.

7 THE UTILITY OF INTERPRETABILITY

Human insight can be crucial to robust solutions that account for gaps and unanticipated changes in the data that drives the behavior of learned control procedures. We discuss several ways in which preserving the ability of expert designers to understand the decision procedure helps.

Tracing the input-output mapping: With concise decision trees, human experts can easily trace the decision process used for particular inputs or sets of inputs. For any input conditions, a path can be traced to a leaf (decision output), and for any output, how it was arrived at can be understood as well. Such tracing can allow easy debugging — "Why were bad outcomes seen for these traces?". This also opens the door to more rigorous analyses of the outcomes for sets of inputs, based on methods like symbolic execution [4].

Identifying potential issues: Experts can often identify overfitting and other problems *a priori* if they understand the procedure, as is the case with the concise decision trees we produce. Our experience itself revealed three such instances:

(1) One feature we encoded for use in our decision trees was a prospective reward from fetching the next chunks at different bitrates. This worked for most videos, giving good average performance. However, for some videos with much higher/lower average bitrate than most other videos, results were inferior. This is due to the reward function using absolute bitrates, and thus not being meaningful across videos. Defining reward in relative terms, *i.e.,* normalized to the maximum possible reward, addresses this issue. A blackbox method, by hiding from human experts the logic used in the rules, makes such improvements more challenging.

(2) We noticed that even after training across the sizable network traces used in past work, our rule sets largely depended on optimistic estimators for throughput, unlikely to work well in more challenging environments, *e.g.,* new geographies a video service expands to where connectivity is more limited and variable. To force more conservative behavior, we can either add such traces to the training data, or restrict the learning approach to use only conservative throughput estimators leading to more stable behavior. Another possibility is to add new features to detect situations where conservative or optimistic behavior would be appropriate. Note that while *given enough appropriate data* blackbox solutions would also potentially overcome such problems, this requires noticing the problem in the first place. Also, such data may not always be available: *e.g.,* if the video service performs poorly in a geography, users may self-select themselves out by dropping the service, thus further skewing the data.

(3) Early in our experiments, we observed a peculiar learned rule that translates to "Never fetch the lowest quality after 45 video chunks." This stemmed from overfitting due to training on one video with 49 chunks (on which most other academic ABR work is also evaluated), where even over a sizable set of traces, typically a large enough buffer was built such that the lowest quality was ruled out for the last few chunks. While this particular artifact would be unlikely to arise in a large provider's pipeline given enough diverse training data, similar problems may occur and go undetected in blackbox methods, especially when the underlying data changes, *e.g.,* if a short-form video service introduces longer videos.

Across these examples, blackboxes can hide problems that might otherwise have been obvious to human experts. Prior work [17] has found problems of this type, *e.g.,* Pensieve, even if conditions are highly favourable, does not always download the last chunk of a video at the highest quality.

Finally, when such problems do present themselves, the recourse with blackboxes, depending on the problem's nature, can involve blindly tinkering with the inputs to the blackbox approach until the outcomes improve, or adding extraneous safeguards for each discovered problem.

7.1 Examining two reconstructions

We next give a view of two reconstructions of different complexity: SRF (simplified, same as in Fig. 9) and Twitch.

Simplified SRF: The output tree reveals an intuitive structure and highlights obvious flaws as we discuss below. (These are only present in the simplification, and not SRF's full tree.)

Fig. 9's caption explains how to read the tree. First it checks the point in playback, concluding that it is in a startup phase *if* playtime is below a threshold. In the startup phase, it checks *if* the possible gain in Reward is large enough to warrant the leveling up by two levels. This is only done *if* we deplete the buffer by not too much when doing so; etc. Of course, behind these loose statements are concrete, parametrized features which describe what the particular throughput estimator is, what reward function is used, etc.

An interesting characteristic of the simplified-SRF tree is that there are no quality changes beyond the startup phase. This is clearly unsuitable in practice, and would be an obvious red flag to any domain expert examining this tree. The full tree does, in fact, use adaptation after startup too, although it is infrequent. We have verified this behavior experimentally as well, where SRF makes frequent quality switches during startup, and much fewer later.

Twitch: Having examined a small simplified tree, we discuss the (full) reconstruction for a more complex target, Twitch.

Twitch's tree visually reveals 3 "branches" of activity, which we call panic, cautious, and upbeat. The panic mode is entered when the throughput is very low. Here the tree is most likely to downgrade the quality by two levels to try to build up buffer, regardless of current buffer occupancy. An example trace captured online shows such behavior at roughly 100 s in playback in Fig. 11.

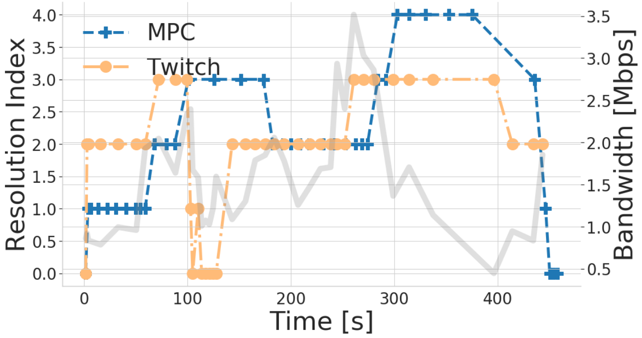


Fig. 11: Twitch shows only marginally more reluctance towards switching when compared to MPC

The cautious mode is entered at mediocre connection quality and, unlike the panic mode, examines the buffer level. In this mode, the most likely action is to either keep the quality or, if buffer-level is low, downgrade it. Downgrading can also be induced by high throughput variance, which indicates uncertain networking conditions.

If throughput is above mediocre, the tree enters the upbeat mode. Here the most common action is upgrading the quality, or if we approach higher quality levels (and therefore, longer download times even with good network conditions), the decision to upgrade is weighted against the buffer drain it would incur, and the current buffer occupancy.

Unlike several other providers, Twitch's reconstruction reveals a willingness to switch qualities often. This is in line with our experimental observation that Twitch and MPC make similar number of switches in the same conditions, while other providers switch much less frequently compared to MPC. Based on this analysis, if a switching-averse provider wanted to adopt Twitch's approach by reconstructing it, they would have to suitably tweak it to reduce switching.

To summarize, with interpretability, we can catch problems before they occur, reason about generalization and behavior across sets of operating conditions instead of just point testing, and methodically discover and fix encountered problems.

8 LIMITATIONS & FUTURE WORK

Over the course of this work, unsurprisingly, we uncovered several shortcomings of our approach, which offer interesting avenues for future exploration:

- Accurate and concise trees require intuitive primitives, *e.g.,* moving averages, which must be manually encoded (§4). Perhaps such primitives can be automatically captured from a corpus of available hand-designed networked algorithms. But this is likely a challenging task.
- We explored a limited set of features, some across only a small part of their possible parameter ranges, *e.g.,* only 5 discrete values for the α parameter in moving averages. A potentially highly effective avenue of improvement lies in tuning the features using a black box optimizer, *e.g.,* a Gaussian Process Optimizer [29], to suggest useful values.
- We can only train for an appropriately specified decision space, as is clear from the failure of our approach for YouTube (§6.2). We can expand the decision space with the benefit of manually-drawn observations from experiments, but automatically discovering it seems difficult.
- We do not expect our approach to always be able to match a target algorithm. However, failures of our approach also help: they often flag "atypical" ABR designs for manual analysis, like for YouTube and Pornhub, and could help uncover unknown (proprietary) insights.
- We used an intuitive but *subjective* definition of "interpretable": trees with under 20 leaves on domain-specific literals. Our own experience with understanding the results was positive, but we hope feedback from other researchers will help sharpen the interpretability goal for future work.
- For providers that customize their ABR for different regions and sets of clients, we can only reconstruct the behavior we observe from our test clients. For future work, this opens an interesting opportunity: observing differently-tuned versions of the same ABR, it may be possible to achieve higher-quality reconstructions, which also identify the parameters whose tuning varies across regions.

9 CONCLUSION

We take the first steps towards an ambitious goal: reconstructing unknown proprietary streaming algorithms in a human-interpretable manner. We customize and evaluate a rule-set approach, achieving good results for reproducing the behavior of algorithms deployed at several popular online services. Our approach produces succinct output open to expert interpretation and modification, and we discuss through several examples, the utility of this interpretability.

While promising, our results also expose a likely fundamental limitation — we need to encode and make available suitable domain knowledge to the learning approach. This can be interpreted as suggesting that we should reconcile learning with our already acquired human expertise, instead of starting afresh. We hope to apply this approach, suitably customized, to congestion control as well, where it is unclear how much diversity there is in actual deployment of different, unknown congestion control algorithms across popular Web services.

Acknowledgments

We are grateful to Martin Vechev, Ce Zhang, our anonymous reviewers, and our shepherd Junchen Jiang, for valuable discussions and feedback. This work is supported by the Swiss National Science Foundation under grant number 182409.

References

[1] Anne Aaron, Zhi Li, Megha Manohara, Jan De Cock, and David Ronca. Per-title encode optimization. https://link.medium.com/jEeb6GV0ZW.

[2] Saamer Akhshabi, Ali Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *ACM MMSys*, 2011.

[3] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: auto-tuning video ABR algorithms to network conditions. In *ACM SIGCOMM*, 2018.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 2018.

[5] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *NeurIPS*, 2018.

[6] Lionel Blondé and Alexandros Kalousis. Sample-efficient imitation learning via generative adversarial nets. In *PMLR*, 2019.

[7] Kyunghyun Cho, B van Merrienboer, Caglar Gulcehre, F Bougares, H Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

[8] Federal Communications Commission. Validated data September 2017 - measuring broadband America. https://www.fcc.gov/reports-research/reports/.

[9] Luca De Cicco and Saveri o Mascolo. An experimental investigation of the Akamai adaptive video streaming. In *USAB*, 2010.

[10] Arnaud Dethise, Marco Canini, and Srikanth Kandula. Cracking open the black box: What observations can tell us about reinforcement learning agents. In *ACM NetAI*, 2019.

[11] Anis Elgabli and Vaneet Aggarwal. Fastscan: Robust low-complexity rate adaptation algorithm for video streaming over HTTP. In *IEEE TCSVT*, 2019.

[12] Maximilian Grüner, Melissa Licciardello, and Ankit Singla. Reconstructing proprietary video streaming algorithms. https://github.com/magruener/reconstructing-proprietary-video-streaming-algorithms, 2020.

[13] H. Guo, Q. Zhang, and A. K. Nandi. Feature generation using genetic programming based on fisher criterion. In *IEEE EUSIPCO*, 2007.

[14] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *NIPS*, 2016.

[15] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *ACM SIGCOMM*, 2014.

[16] Ayad Ibrahim, Im Youngbin, Keller Eric, and Ha Sangtae. A practical evaluation of rate adaptation algorithms in HTTP-based adaptive streaming. In *Computer Networks*, 2018.

[17] Yafim Kazak, Clark Barrett, Guy Katz, and Michael Schapira. Verifying deep-RL-driven systems. In *ACM NetAI*, 2019.

[18] Breiman L., Friedman J. H., Olshen R. A., and Stone C. J. *Classification and Regression Trees*. Wadsworth and Brooks, 1984.

[19] De Cicco L. and Mascolo S. A mathematical model of the Skype VoIP congestion control algorithm. In *IEEE Transactions on Automatic Control*, 2010.

[20] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward a practical perceptual video quality metric. https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652, 2016.

[21] Melissa Licciardello, Maximilian Grüner, and Ankit Singla. Understanding video streaming algorithms in the wild. In *PAM*, 2020.

[22] F. T. Liu, K. M. Ting, and Z. Zhou. Isolation forest. In *IEEE ICDM*, 2008.

[23] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *ACM SIGCOMM*, 2017.

[24] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. https://github.com/hongzimao/pensieve, 2017.

[25] Zili Meng, Jing Chen, Yaning Guo, Chen Sun, Hongxin Hu, and Mingwei Xu. PiTree: Practical implementation of ABR algorithms using decision trees. In *ACM Multimedia*, 2019.

[26] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Explaining deep learning-based networked systems. *arXiv:1910.03835*, 2019.

[27] Abhijit Mondal, Satadal Sengupta, Bachu Rikith Reddy, M. J.V. Koundinya, Chander Govindarajan, Pradipta De, Niloy Ganguly, and Sandip Chakraborty. Candid with YouTube: Adaptive streaming behavior and implications on data consumption. In *ACM NOSSDAV*, 2017.

[28] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video QoE fairness. In *ACM SIGCOMM*, 2019.

[29] Michael Osborne, Roman Garnett, and Stephen Roberts. Gaussian processes for global optimization. In *International Conference on Learning and Intelligent Optimization*, 2009.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[31] Yanyuan Qin, Shuai Hao, Krishna R Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *ACM CoNEXT*, 2018.

[32] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3G networks: analysis and applications. In *MMSys*, 2013.

[33] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE*, 2017.

[34] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *PMLR*, 2011.

[35] Sandvine. The global Internet phenomena report. https://www.sandvine.com/press-releases/sandvine-releases-2019-global-internet-phenomena-report, 2019.

[36] Reliable Secure and Intelligent Systems Lab. JSNice - statistical renaming, type inference and deobfuscation. http://jsnice.org/, 2018.

[37] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the DASH reference player. In *ACM MMSys*, 2018.

[38] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. In *IEEE INFOCOM*, 2016.

[39] Trevor Stephens. Genetic programming in Python. https://github.com/trevorstephens/gplearn, 2017.

[40] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck. HTTP/2-based adaptive streaming of HEVC video over 4G/LTE networks. In *IEEE Communications Letters*, 2016.

[41] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *PMLR*, 2018.

[42] Ruohan Wang, Carlo Ciliberto, Pierluigi Vito Amadori, and Yiannis Demiris. Random expert distillation: Imitation learning via expert policy support estimation. In *PMLR*, 2019.

[43] Shichang Xu, Subhabrata Sen, and Z. Morley Mao. CSI: Inferring mobile ABR video adaptation behavior under HTTPS and QUIC. In *ACM EuroSys*, 2020.

[44] Y. Xu, C. Yu, J. Li, and Y. Liu. Video telephony for end-consumers: Measurement study of Google+, iChat, and Skype. In *IEEE/ACM Transactions on Networking*, 2013.

[45] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware Internet video delivery. In *USENIX OSDI*, 2018.

[46] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *ACM SIGCOMM*, 2015.

Midgress-aware traffic provisioning for content delivery

Aditya Sundarrajan
UMass Amherst

Mangesh Kasbekar
Akamai Technologies

Ramesh K. Sitaraman
UMass Amherst
& Akamai Technologies

Samta Shukla
CVS Health

Abstract

Content delivery networks (CDNs) cache and deliver hundreds of trillions of user requests each day from hundreds of thousands of servers around the world. The traffic served by CDNs can be partitioned into hundreds of traffic classes, each with different user access patterns, popularity distributions, object sizes, and performance requirements. Midgress is the cache miss traffic between the CDN's servers and the content provider origins. A major goal of a CDN is to minimize its midgress, since higher midgress translates to higher bandwidth costs and increased user-perceived latency.

We propose algorithms that provision traffic classes to servers such that midgress is minimized. Using extensive traces from Akamai's CDN, we show that our midgress-aware traffic provisioning schemes can reduce midgress by nearly 20% in comparison with the midgress-unaware schemes currently in use. We also propose an efficient heuristic for traffic provisioning that achieves near-optimal midgress and is suitable for use in production settings. Further, we show how our algorithms can be extended to other settings that require minimum caching performance per traffic class and minimum content duplication for fault tolerance. Finally, our paper provides a strong case for implementing midgress-aware traffic provisioning in production CDNs.

1 Introduction

Content delivery networks (CDNs) carry more than 50% of all Internet traffic today [35] and that fraction is projected to increase over the coming years. Modern CDNs host a wide variety of content such as videos, software downloads, web pages, etc. that belong to hundreds of content providers. CDNs deploy hundreds of thousands of servers in clusters at the edge of the Internet to serve the hosted content to billions of end-users around the world. If the requested content is available in the edge server, a cache hit occurs and the end-user experiences a quicker response with lower latency. Otherwise, a cache miss occurs, and the edge server must fetch the content from the content provider's origin. A cache miss increases the user-perceived latency for a response and also increases the "midgress" traffic, which is the cache miss traffic between the CDN's edge servers and the content provider origins.

A CDN has many performance and cost objectives that must be optimized. Three important metrics are *origin offload* that is the amount of traffic offloaded from the origin servers, *end-user latency* that is the time between request and response for content as perceived by the end-user, and the *midgress bandwidth cost*[1] that is the cost of internal traffic in the CDN caused mainly due to cache misses at the edge servers. A metric that ties the three objectives together is the cache miss rate[2] which is the fraction of content bytes that were not present in the edge caches and needed to be fetched from origin. Smaller miss rate implies lesser cache miss traffic. Reduced cache miss traffic in turn implies increased origin offload, reduced end-user latency and reduced midgress bandwidth cost. Hence, minimizing the midgress, is a key performance objective from multiple perspectives.

Traffic classes. When users request content that is hosted on a CDN, the requests are classified into *traffic classes*. A traffic class is a collection of domains that host a specific type of content belonging to one or more content providers with similar requirements. For example, CNN videos and Apple iOS software downloads are each examples of a traffic class. Large CDNs host content that belong to hundreds of traffic classes. Recent work [66] has shown that traffic classes hosted on CDNs exhibit wide variations in popularity distributions, object size distributions and caching characteristics.

How CDNs serve content to users. Two interacting systems determine how content is served to users.

1) The *traffic provisioning system* decides which servers serve what fraction of each traffic class. Traffic provisioning

[1] The CDN also incurs a bandwidth cost for the "egress" traffic of content sent from the edge servers to the end-users. However, content providers pay the CDN for their egress traffic, while the midgress traffic is purely an overhead for the CDN operator that must be minimized.

[2] The miss rate metric that we use in this paper is sometimes called *byte* miss rate. An alternate definition is the (unweighted) fraction of *requests* that are cache misses and is less relevant for our work.

is performed periodically (say, once every few hours) as an *offline process* and uses the predicted user demand for the traffic classes and available server resources to produce an assignment of traffic classes to servers. Subsequently, each user request of each traffic class is routed [12] in *real-time* to a server that is provisioned to serve that traffic class[3].

2) Each CDN server has a cache that stores the content requested by users. Each server employs a *cache management system* that implements policies for managing the cache, such as an admission policy to decide what objects are cached and an eviction policy to decide what objects are evicted.

Minimizing midgress. The midgress bandwidth could cost tens of millions of dollars a year[4]. Thus, even a small reduction in midgress can be significant. Much of the prior work has focused on better cache management for reducing cache misses The past decades have seen research on numerous caching algorithms, such as Adapt-Size [4], Cliffhanger [15], SLRU [40], TLRU [23], S4LRU [34], CFLRU [59], ARC [53], LRU-S [65], LRU-K [56], and GDS [7]. However, the *complementary* problem of optimizing the traffic provisioning process to minimize midgress has not received much attention. In the current state-of-the-art, production CDNs assign traffic classes to servers with the goal of not overloading the servers, without explicitly minimizing midgress.

Our work shows that traffic provisioning in a midgress-aware manner can provide additional benefits to what can accrue from better cache management alone. Our traffic provisioning approach incorporates *both* traditional load balancing and the newer midgress considerations to minimize midgress traffic. *The main thesis of the paper is that by explicitly incorporating midgress considerations, it is possible to devise traffic provisioning schemes that minimize midgress traffic by nearly 20%, potentially resulting in millions of dollars of bandwidth cost savings.* Further, the midgress reduction due to better traffic provisioning is *complementary* to any improvements in cache management. As CDNs already implement traffic provisioning algorithms, albeit in a midgress-unaware manner, our contribution can be viewed as a drop-in replacement for an existing (midgress-unaware) traffic provisioning system.

Why be midgress-aware? "Midgress-aware" traffic provisioning algorithms explicitly incorporate cache miss traffic in addition to "balancing" the load. We illustrate the need for midgress awareness through a simple example. Consider two servers and three traffic classes. Each server has a cache size of 4 TB and sufficient capacity to serve all traffic classes. The three traffic classes have equal load of λ that need to be assigned to the two servers. The miss rate curves (MRCs) for the three traffic classes are as shown in Figure 1. The MRCs of traffic classes TC_1 and TC_3 flatten out quickly. This means that they require very little cache space to achieve the best possible performance. On the other hand, traffic class TC_2 has a slowly decreasing gradient. Thus, the miss rate of TC_2 keeps decreasing as more cache space is allocated to it.

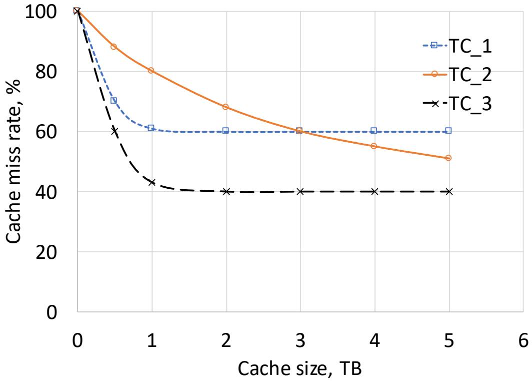


Figure 1: MRCs of traffic classes TC_1, TC_2 and TC_3.

Current traffic provisioning algorithms are *midgress-unaware* in that they only ensure that no server is overloaded. Such an algorithm could choose *any* assignment of traffic classes to servers, since any server has sufficient capacity to serve all classes, e.g., assigning TC_1 and TC_2 to server 1 and TC_3 to server 2 is one possible solution. More generally, any assignment with $(x+y+z)\times\lambda$ traffic to server 1 and $((1-x)+(1-y)+(1-x))\times\lambda$ traffic to server 2 is feasible, where x,y and $z\in[0,1]$, are the traffic fractions of TC_1, TC_2 and TC_3 respectively. Note that in this paper, we split the load of a traffic class by requests.

On the other hand, a midgress-aware algorithm would choose an assignment that minimizes the overall cache miss traffic from the two servers, while also ensuring that no server is overloaded. In the above example, assigning all of TC_1 and TC_3 to server 1 and all of TC_2 to server 2 would result in the least amount of cache miss traffic from the two servers. This is because TC_2 gets the largest cache space possible for its entire load and TC_1 and TC_3 get enough space to achieve the smallest cache miss rates.

1.1 Contributions

We make the following contributions.

1) We develop an optimization model for midgress-aware traffic provisioning that assigns traffic classes to servers in a manner that minimizes midgress traffic. The model is a non-convex mixed-integer linear program (MILP) that we solve using CPLEX. Our work is the first to explicitly model and minimize midgress in the traffic provisioning process. *Since a large CDN could incur a midgress of 10+ Tbps at a cost of $60+ million/year, even a small midgress reduction translates into large cost savings for the CDN.*

[3] The results of the traffic provisioning are used to create DNS records that can be resolved by the user in real-time using a DNS lookup [12].

[4] As a back-of-the-envelope calculation, a large CDN serving 50 Tbps of egress traffic at a 20% miss rate at the edge has a midgress traffic of 10 Tbps. The price of network bandwidth varies greatly throughout out the world. Though hard to estimate accurately, assuming a blended price of 50 cents per Mbps per month, midgress bandwidth costs 60 million dollars per year.

2) We apply our optimization solution to *metro-level traffic provisioning* where the traffic classes provisioned to server clusters within a metro area (e.g., NY city) are re-provisioned to minimize midgress. Metro-level traffic re-provisioning is a common operation, since the latency impact of moving a traffic classes across clusters within the same metro is likely minimal. Using extensive production traces from Akamai, we show that our midgress-aware traffic provisioning can reduce the midgress of a metro-area by 18.37% on average compared to midgress-unaware provisioning.

3) We also use our optimization solution for *cluster-level provisioning* where the traffic classes assigned to servers within a cluster are re-provisioned to minimize midgress. Cluster-level traffic (re-)provisioning is also a common operation since moving a traffic class across servers within the same cluster will likely not impact end-user latencies. Using production traces from Akamai's CDN, we show that cluster-level provisioning in conjunction with metro-level provisioning can reduce the midgress of a traffic class by 41.07% on average compared to midgress-unaware provisioning.

4) To be useful in practice, midgress-aware traffic provisioning has to be computationally efficient. We propose a midgress-aware heuristic called `local search` that is fast and near-optimal. The midgress achieved by `local search` was within 1.1% of optimal for both the metro-level and the cluster-level traffic provisioning. Further, in our experiments, `local search` completed in only 2 minutes, while finding the optimal took several hours.

5) We also show that our traffic provisioning algorithms are robust across different cache management policies and provide a midgress reduction in the range of 7.76% - 13.3%.

6) CDN operators often have to deal with additional constraints such as maintaining a certain level of traffic class redundancy or guaranteeing a minimum level of caching performance for traffic classes. We show how the optimization model for midgress-aware traffic provisioning and the heuristic algorithm, `local search`, can be extended to accommodate such constraints.

7) While the above results are for "shared" caches where a single unpartitioned cache is used to store objects from all traffic classes, we show that our traffic provisioning approach can be modified to work with "partitioned" caches where each traffic class is assigned a separate cache partition. We show that the midgress of partitioned caches can be reduced by more than 14% using our midgress-aware traffic provisioning approach, when compared to a midgress-unaware baseline.

1.2 Roadmap

The rest of the paper is organized as follows. In Section 2, we model midgress-aware traffic provisioning as a non-convex mixed-integer optimization problem. In Section 3, we propose a faster heuristic for midgress-aware traffic provisioning called `local search`, as well as a midgress-unaware baseline called `baseline fit`. In Section 4, we evaluate our optimization model and heuristics using extensive traces from Akamai's production CDN to empirically understand the midgress reduction achieved by our algorithms. In Section 5, we extend and evaluate our midgress-aware traffic provisioning algorithms to include other constraints such as minimum redundancy and maximum cache miss rates. Further, we extend our work to partitioned caches. We discuss some related work in Section 6 and conclude in Section 7.

2 Optimization model for traffic provisioning

We model traffic provisioning in a CDN as follows. We are given a set of N traffic classes. For each traffic class j, we are given the (predicted) amount of load of λ_j Gbps, $\forall j \in 1 \ldots N$. The predicted load for traffic provisioning is derived from historical load values for these classes by the CDN. Further, we are given M sites where the i^{th} site has a cache of size C_i TB and a capacity of T_i Gbps, $\forall i \in 1 \ldots M$. In cluster-level traffic provisioning, each site models a single CDN server within a cluster of M servers. In the more complex setting of metro-level traffic provisioning, we model an *entire* cluster as a single site within a metro area with M clusters. While not strictly accurate, we show that viewing the entire cluster as a single site in the metro-area setting is useful in practice. The capacity (resp. cache size) of each site is calculated as either the capacity (resp. cache size) of a single server in the former setting or as the aggregate capacity (resp. cache size) of the entire cluster in the latter setting. Henceforth, a site refers to a server in the cluster-level setting and a cluster in the metro-level setting.

The goal of traffic provisioning is to produce an assignment of traffic classes to sites, such that the total midgress across all the sites is minimized within the constraint that no site is assigned more load than its capacity. Note that a traffic class may be fractionally assigned across multiple sites, e.g., a traffic class with 10 Gbps of load can be assigned across two sites to host 7 Gpbs and 3 Gbps each of that class[5].

2.1 Modeling cache eviction and midgress

Given a site with an assignment of traffic classes, we need to model the miss traffic (i.e., midgress) that will result from serving those classes. The miss traffic is dependent on the cache management policies used by the sites. Nearly all production CDN caches use LRU (least-recently-used) variants as their eviction policy, since it is very efficient and achieves a comparable (byte) miss rate for typical CDN content traffic in comparison with other more complex eviction policies. For example, Akamai servers evict content using LRU, while admitting objects on second hit [47]. Production installations of the popular content caches Varnish [39] and NGINX [63]

[5] A CDN can implement such a fractionally-provisioned traffic class via a DNS mechanism that returns the ip address of the first site 70% and ip address of the second site 30% of the time.

also use LRU variants, as do recent academic work on content caching such as AdaptSize [4].

Production CDN servers also typically use a shared cache architecture where each server uses a single unpartitioned cache to serve all its traffic classes [66]. It is known that a partitioned cache that is sized in an optimal fashion can yield a greater reduction in midgress over a shared unpartitioned cache under the independent reference model (IRM) traffic assumptions [20]. However, in a production CDN, each server hosts a large number of traffic classes. Further, both the set of traffic classes hosted by a given server and the volume of traffic served per class by that server varies throughout the day. Thus, there is *significant* overhead involved in maintaining multiple cache partitions whose sizes must be dynamically varied throughout the day. The constant resizing of cache partitions could itself also lead to an increase in the midgress [61]. For these reasons, a shared unpartitioned cache is typically used by CDNs in practice.

In light of the above discussion, since our goal is to devise traffic provisioning algorithms to reduce midgress in production CDN settings, we develop a model for sites that use an LRU cache eviction policy with a shared cache architecture. But, later, we show empirically that our optimization model and algorithms produce a significant reduction in midgress, even if the CDN were to use other eviction policies (Section 4.3). Further, we show that our approach can also be easily extended to provide midgress reduction in a partitioned cache architecture (Section 5.3).

Eviction age equality. The eviction age of an object in cache is the difference between the time the object is evicted and the time that it was last accessed. In an LRU cache, at the time of access, the object goes to the head of the LRU list. Then, the eviction age of the object is the time for that object to move from the head to the tail of the LRU list and then get evicted. Thus, this time is about the same for all objects, when the size of an object is small with respect to the size of the cache. *We make the modeling assumption that the eviction age of all objects in cache are equal.* This assumption is also borne out in production caches and the common eviction age of the objects is logged as the eviction age of the cache.

The notion of eviction age can be extended to a traffic class by averaging the eviction age of all the requested objects from that traffic class. Since we model each object as having the same eviction age, all traffic classes assigned to a site share the same cache, and so they must have the same eviction age, which we also denote to be the eviction age of the cache. The eviction age of a cache has a direct relationship with the cache hit rate. Requests that have inter-arrival times less than or equal to the eviction age experience a cache hit and the rest experience a cache miss. So, for a given mix of traffic classes, as the cache size increases, the eviction age increases and so does the cache hit rate. Eviction age of a cache is similar to the concept of window size in [24]. Eviction age equality is crucial in our modeling of the midgress of traffic classes that share a single LRU cache.

2.2 Formulation of our optimization model

We now formulate our optimization model (referred to as OPT henceforth) for midgress-aware traffic provisioning.

Inputs of OPT. The input parameters used in the model are summarized in Table 1. We are given N traffic classes and M sites. The load λ_j of the j^{th} traffic class is given, for all $1 \leq j \leq N$. The cache size C_i and the capacity T_i of the i^{th} site is also given, for all $1 \leq i \leq M$. Further, for each traffic class, we are given the miss rate curve (MRC) and eviction age function as described below.

1) Miss rate curve (MRC), $\mathcal{M}_j(c)$. The MRC of a traffic class plots the cache miss rate as a function of cache size c. In this work, we assume that this function is convex (decreasing) which is generally true for stack-based algorithms [66]. As examples, MRC of two traffic classes, traffic class 2 and 14 (see Table 3) are shown in Figure 2.

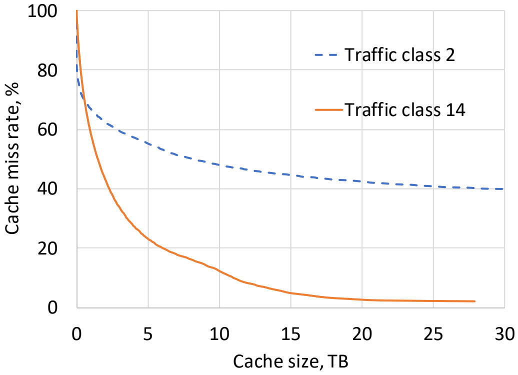


Figure 2: MRCs of two traffic classes.

From Figure 2, we can see that the MRCs are both convex. However, their gradients vary at different rates. Traffic class 2 has higher gradient at very small cache sizes but gradually flattens out as it reaches a cache space of 30 TB. Traffic class 14 on the other hand has a relatively high gradient until about 15 TB after which the MRC flattens out.

2) Eviction age function, $\mathcal{T}_j(c,\lambda)$. The eviction age function of a traffic class plots the eviction age at load λ as a function of the cache size c. The eviction age function also gives us information about *footprint pressure* of a traffic class, which is a relative measure of the amount of unique bytes accessed over a time period. A traffic class has high footprint pressure if a large number of unique bytes are accessed over a short time period. In this work, we assume that the eviction age function is convex (increasing) based on observations from production traces. The eviction age functions of two traffic classes, 2 and 14 (see Table 3) are shown in Figure 3.

From Figure 3, we can see that the eviction age functions are convex. As expected, at the given load, the eviction age increases with increase in cache size. Note that until about an eviction age of 2.1 days, traffic class 14 has higher footprint

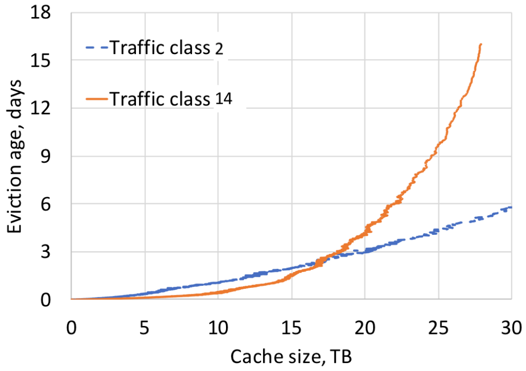


Figure 3: Eviction age functions of two traffic classes.

pressure when compared to traffic class 2, after which this behavior is flipped. Hence, if traffic classes 2 and 14 are assigned to the same site, traffic class 14 gets more cache space at smaller eviction ages ($\leq$ 2.1 days) due to higher footprint pressure and lesser cache space at larger eviction ages ($>$ 2.1 days) due to smaller footprint pressure.

To efficiently compute the MRC and eviction age function for every traffic class, we use a succinct space-time representation of the cacheability properties of a traffic class known as footprint descriptors [66].

We now briefly describe footprint descriptors.

Footprint descriptors. A footprint descriptor is a space-time representation of the caching properties of a traffic class. It is the joint probability distribution of the stack distance [51] (aka reuse distance) and the interarrival time distributions of a traffic class. A footprint descriptor can be used to determine the cache size and the eviction age that is required to achieve a certain cache hit rate, as a function of the traffic load. A footprint descriptor can also be used to determine the eviction age of a cache at different cache sizes and vice versa.

A reuse sequence is a sequence of requests where the first and the last request in the sequence is for the same object and that object is not requested anywhere else in that sequence. A simplified version of a footprint descriptor of a traffic class j is a tuple $<\lambda_j, P^r(s,t))>$ where λ_j is the load of traffic class j and $P^r(s,t)$ is the reuse-sequence descriptor which is the joint probability distribution that a reuse sequence of the traffic class has s unique bytes and time duration t. Given a footprint descriptor, the miss rate curve at cache size s is defined as $MRC(s) = 1 - \sum_{s' \leq s} \sum_{t} P^r(s',t)$. The eviction age function $\mathcal{T}_j(s,\lambda_j)$, is computed from $P^r(s,t)$ by plotting the duration t as a function of unique bytes s at load λ_j.

Given the footprint descriptor of different traffic classes, the footprint descriptor of a traffic mix can be computed using the addition operator ($\oplus$) of the footprint descriptor calculus [66]. The crux of the addition operation is the convolution of the joint probability distribution, $P^r(s,t)$, of all the traffic classes, which can be efficiently computed using the Fast Fourier Transform algorithm. The MRC of the traffic mix can then be computed from the footprint descriptor of the traffic mix as described above. Note that the request characteristics of a traffic class could change slowly over time, requiring the footprint descriptor to be recomputed periodically.

Outputs of OPT. The output parameters of OPT are presented in Table 2. The primary output is x_{ij} that represents the fraction of traffic class j assigned to site i.

Notation	Description
N	Number of traffic classes
M	Number of sites
λ_j	Load of traffic class j
$\mathcal{M}_j(c_{ij})$	Miss rate of traffic class j at cache capacity c_{ij} in site i
$\mathcal{T}_j(c_{ij},\lambda_j)$	Eviction age of traffic class j at cache capacity c_{ij} and load λ_j in site i
C_i	Cache size of site i
T_i	Capacity of site i

Table 1: Input parameters of optimization model.

Notation	Description
c_{ij}	Cache size occupied by traffic class j in site i
ρ_i	Eviction age of site i and of traffic classes assigned to site i
x_{ij}	Fraction of $\lambda_j \in [0,1]$ assigned to site i

Table 2: Output parameters of optimization model.

Objective function. The objective of midgress-aware traffic provisioning is to assign the N traffic classes to the M sites such that the midgress traffic from all the sites is minimized as follows.

$$\min. \sum_{i=1}^{M}\sum_{j=1}^{N} x_{ij}\lambda_j \mathcal{M}_j(c_{ij}) \tag{1}$$

Resource constraints. The first set of constraints are the cache size and the capacity constraints of each site.

$$\sum_{j=1}^{N} c_{ij} \leq C_i \quad \forall i = 1 \dots M \tag{2}$$

$$\sum_{j=1}^{N} x_{ij}\lambda_j \leq T_i \quad \forall i = 1 \dots M \tag{3}$$

The cache size constraint (Equation 2) states that the cache size occupied by all traffic classes assigned to all sites must not exceed the cache size of the site. The capacity constraint (Equation 3) states that the load of all traffic classes assigned to all sites should not exceed the capacity of the site.

Eviction age equality constraint. The eviction age function, $\mathcal{T}_j(c_{ij},\lambda_j)$ is defined at load λ_j for traffic class j. When traffic class j is assigned to site i, its load can be less than or equal to λ_j due to fractional assignments. Let the load of traffic class j assigned to site i be $\lambda'_j \leq \lambda_j$. Then, the eviction age of traffic class j in site i is.

$$\mathcal{T}_j(c_{ij},\lambda'_j) = \frac{\mathcal{T}_j(c_{ij},\lambda_j)}{\lambda'_j/\lambda_j} = \frac{\mathcal{T}_j(c_{ij},\lambda_j)}{x_{ij}} = \rho_i$$

The first equality is due to the fact that decreasing the load of a traffic class by a factor increases the eviction age of that class by the same factor, since eviction rate decreases by that factor. In the last equality, ρ_i is the eviction age of site i which is also the eviction age of all traffic classes that are assigned to site i. The eviction age equality constraint for all traffic classes at all sites is then given by

$$\mathcal{T}_j(c_{ij},\lambda_j) = \rho_i x_{ij} \quad \forall j(i) = 1 \ldots N(M). \tag{4}$$

As previously discussed, the eviction age equality constraint in Equation 4 establishes the condition under which traffic classes assigned to site i share the cache.

Load assignment constraint. The load of a given traffic class can be fractionally assigned across sites. This means that for some traffic class j, 50% of the load λ_j could be assigned to site 1, 30% to site 2 and the remaining 20% to site 3, and so on. The load assignment constraint ensures that all the load of each traffic class is assigned to one or more sites.

$$\sum_{i=1}^{M} x_{ij} = 1 \quad \forall j = 1 \ldots N \tag{5}$$

Non-negativity constraints. The output parameters ρ_i, c_{ij} and x_{ij} should be non-negative.

$$\rho_i > 0 \quad \forall i = 1 \ldots M \tag{6}$$

$$c_{ij} \geq 0 \quad \forall j = 1 \ldots N \tag{7}$$

$$x_{ij} \in [0,1] \quad \forall j(i) = 1 \ldots N(M) \tag{8}$$

Together, Equations 1-8 constitute the optimization model for midgress-aware traffic provisioning OPT.

2.3 Solving the optimization model OPT

The complexity of solving the optimization model OPT proposed in Section 2.2 is evaluated as follows. The objective function (Equation 1) is biconvex since the load fraction x_{ij} is linear and the MRC $\mathcal{M}_j(c_{ij})$ is convex. Equations 2-3, 5-8 are affine constraints. The eviction age function $\mathcal{T}_j(c_{ij})$ is convex and the product term $\rho_i x_{ij}$ is bilinear. Equation 4 is a non-convex constraint because the feasible set defined by this constraint is non-convex. Overall, the optimization problem is non-convex and in general an NP-hard problem. We make a number of mathematical transformations to convert the optimization problem to a mixed integer linear program (MILP), which in turn can be solved efficiently using CPLEX.

3 Traffic provisioning heuristics

The optimization model OPT proposed in Section 2.2 is an NP-hard problem and it can take several hours for a solver to obtain the exact optimal solution. A faster but approximate solution is valuable for a large production CDN that has hundreds of traffic classes, 1000+ clusters with deployments in every major metro region of the world. To that end, we propose a traffic provisioning heuristic called `local search` that is fast and sufficiently accurate to be used in production. Intuitively, our traffic provisioning heuristic is a “hill climbing” solution for our optimization model in Section 2.2. We also consider a midgress-unaware traffic provisioning algorithm called `baseline fit` that we use as a baseline to evaluate the benefits of being midgress-aware. The `baseline fit` algorithm is similar to the midgress-unaware algorithms currently used in production settings.

3.1 Midgress-unaware baseline

The midgress-unaware traffic provisioning algorithm called `baseline fit` (see Algorithm 1) is based on consistent hashing, similar to the algorithms used in production settings [47]. The algorithm takes as input the set of N traffic classes and the set of M sites that are both hashed to points on a unit circle. The traffic classes are picked in a random order and assigned to sites as follows. Each traffic class j is assigned to the nearest site i on the unit circle in the clockwise direction. If the chosen site i does not have enough capacity to host the entire load λ_j, then a first fit algorithm is used, starting from the chosen site i, and continuing to subsequent sites on the unit circle in the clockwise direction, until all traffic is assigned. The key point to note is that `baseline fit` does not explicitly minimize the miss traffic, but rather it only ensures that no site gets more load than its capacity. That is, it produces a *feasible* solution for our model OPT by obeying Equations 2 - 8, but does not minimize midgress.

Algorithm 1 `Baseline fit` algorithm

```
Input: $N, M, \lambda_j, C_i, T_i$
Output: Fraction of traffic class $j$ assigned to site $i$, $x_{ij}, \forall j(i) = 1 \ldots N(M)$
$x_{ij} = 0$
$TC_{set}$ = set of all traffic classes arranged in random order
$S_{set}$ = set of all sites hashed to a unit circle
for all $j \in TC_{set}$ do
    $i$ = Site chosen by consistent hashing
    if site $i$ has remaining traffic capacity $\geq \lambda_j$ then
        $x_{ij} += 1$
    else
        Assign traffic fraction $\lambda_j$ using first fit starting from site $i$ on the unit circle
```

3.2 Midgress-aware local search

We propose a midgress-aware traffic provisioning algorithm called `local search` (see Algorithm 2) that uses a hill climbing approach to solve the optimization model OPT. It is designed to be fast, but may not always produce the optimal solution. The algorithm `local search` begins with a feasible assignment as determined by `baseline fit`. The algorithm operates in rounds where every traffic class is picked one at a time in each round. The traffic class that is picked is reassigned in small increments of a fraction δ ($0 < \delta < 1$) of its load to the server that minimizes the midgress objective while maintaining feasibility. If a round does not decrease the midgress traffic objective by at least a specified $\varepsilon << 1$, the algorithm stops and outputs the final assignment.

Note that `local search` could end up in a local optimum that isn't close to the global optimum. However, `local search` is efficient enough that it can be run multiple times (in parallel) with different starting points to improve a suboptimal solution. We discuss the running time of `local search` in Section 4.

Computing the midgress of a traffic assignment. The `local search` algorithm requires an efficient way to compute the midgress traffic of each site, given a traffic class assignment. A known technique for computing miss traffic of a site is footprint descriptor calculus [66]. Knowing the footprint descriptor of each traffic class that is assigned to a site, we use the calculus to efficiently derive the footprint descriptor for the traffic mix, that in turn provides the MRC of the traffic mix, from which we derive the midgress of the traffic mix.

4 Experimental evaluation

Using production traces collected from a metro area of Akamai's CDN, we compare the midgress of OPT with that of `baseline fit` and `local search` in both metro-level and cluster-level traffic provisioning. We perform the evaluation in two steps: 1) We evaluate metro-level traffic provisioning by viewing each cluster as a site. The site is assumed to have cache size and capacity equal to the sum of the cache sizes and capacities of all servers in that cluster. The output of metro-level traffic provisioning is an assignment of traffic classes to clusters that minimizes the midgress of the metro area. 2) The output of metro-level traffic provisioning is the input to cluster-level traffic provisioning. We evaluate cluster-level traffic provisioning by assigning traffic classes to servers within a cluster to further minimize midgress.

Production traces. To perform our evaluation, we collect production traces from all Akamai CDN servers from a metro area serving traffic for 25 traffic classes over a period of 16 days. The characteristics of the traffic classes are listed in Table 3. From Table 3, we see that, in this metro area, 9 traffic classes serve web content, 11 traffic classes serve media content and the remaining 5 traffic classes serve software downloads. The traffic classes exhibit a wide variation in load (Gbps), arrival rate (requests/sec), content footprint (in unique bytes), and number of objects. The majority of the load is for media content at 47.3% followed by software downloads at 41.5% and web content at 11.2%. In terms of the unique bytes that are cached in the metro area, the majority is again for media content at 60.9%, followed by 25.6% for web content and 13.5% for software downloads.

Algorithm 2 `Local search` algorithm

```
Input: $N, M, \lambda_j, C_i, T_i$
Output: Fraction of traffic class $j$ assigned to site $i$, $x_{ij}, \forall j(i) = 1 \dots N(M)$
Get feasible assignment using baseline fit algorithm
$TC_{set}$ = set of all traffic classes arranged in random order
$S_{set}$ = set of all sites
while True do
    $mg_{curr}$ = midgress of current assignment
    for all $j \in TC_{set}$ do
        $x_{ij} = 0 \quad \forall i = 1 \dots M$
        $\lambda' = \lambda_j$
        while $\lambda' > 0$ do
            $S^j_{set} \subseteq S_{set}$ = set of all sites with remaining traffic capacity $\geq \delta\lambda_j$
            if $S^j_{set} \neq \emptyset$ then
                $i$ = site in $S^j_{set}$ that gives the lowest overall midgress after assigning TC $j$
                $x_{ij}$ += $\delta$
            else
                Assign load $\delta\lambda_j$ using fractional first fit starting from a random site
            $\lambda'$ −= $\delta$
    $mg_{new}$ = midgress of new assignment
    if $mg_{curr} - mg_{new} < \varepsilon$ then
        break
```

Footprint descriptors described in [66] are periodically computed for all traffic classes on the production CDN. We use these footprint descriptors to compute the MRCs and the eviction age functions for the 25 traffic classes in Table 3, to be used as inputs to our traffic provisioning algorithms.

Evaluation setup. To evaluate the traffic provisioning algorithms, we simulate a small metro region with 10 clusters, each containing 10 servers[6]. The capacity of the metro region is set so that the average load is 70% of capacity to reflect the load-to-capacity ratio in a typical CDN. We evaluate the traffic provisioning algorithms at different cache sizes per cluster of 1 TB, 5 TB, 10 TB, 20 TB, 40 TB and 50 TB. For simplicity, we assume that every cluster in the metro area has equal capacity and cache size. Every server within a cluster is also assumed to have equal capacity and cache size.

OPT is solved using CPLEX as part of the GAMS modeling

[6] While a metro region in a large CDN typically has much larger server deployments, we simulate a scaled-down version to keep our experiments computationally tractable.

Traffic class id	Content type	Load (Gbps)	Arrival rate (req/s)	Unique bytes (TB)	Unique objects (million)
1	web	0.39	438.41	1.83	16.36
2	web	1.12	232.48	70.74	38.38
3	media	3.75	345.94	198.85	176.68
4	web	0.24	143.67	0.008	0.03
5	web	0.17	145.13	0.03	0.08
6	download	4.74	1338.91	28.16	19.55
7	web	0.30	851.73	6.21	70.23
8	web	0.58	1213.87	6.38	137.60
9	web	1.59	714.42	22.58	52.91
10	download	0.39	307.92	1.68	0.82
11	download	10.66	809.29	22.74	10.75
12	media	0.43	110.22	14.13	24.41
13	web	0.0013	136.32	0.04	3.58
14	media	7.54	93.01	30.55	2.90
15	media	7.22	89.28	30.14	2.86
16	media	6.04	75.14	30.38	2.89
17	media	0.37	139.23	12.41	26.59
18	web	2.12	935.76	83.42	93.54
19	media	0.35	134.87	24.48	25.12
20	download	1.36	276.63	3.12	2.07
21	media	0.08	9.94	7.31	6.43
22	media	0.90	214.53	43.48	77.90
23	media	0.44	48.28	28.53	26.83
24	media	0.38	78.09	35.25	55.06
25	download	6.99	1879.65	44.94	21.02

Table 3: Traffic class characteristics

language. We use a macOS machine with a 3 GHz Intel Xeon processor with 10 cores and 128 GB RAM for all our experiments. The GAMS program is set to run in parallel mode using 20 threads with a relative optimality gap of 1e-9 and a maximum run time of 40,000 s. Given the complexity of the optimization model, the GAMS program almost always runs for 40,000 s. At that point, the solver has converged to a solution that seldom changes and achieves a relative gap of under 5% at smaller cluster cache sizes less than or equal to 10TB and a relative gap of under 10% at larger cache sizes. A single run of `baseline fit` takes about 1 s and a single run of `local search` takes about 120 s.

4.1 Metro-level traffic provisioning

We evaluate OPT, `baseline fit`, and `local search` by computing the cache miss rate of the entire 10-cluster metro area for different cache sizes. These three algorithms each assign the set of 25 input traffic classes to the clusters in the metro. In the case of `baseline fit` and `local search`, we report the average cache miss rate of 100 runs, where each run considers the traffic classes in a random order. The 95% confidence intervals of the expected cache miss rates have a margin of error of less than 0.4%.

From Figure 4 we can see that OPT gives a 18.37% reduction in midgress on average compared to the midgress-unaware `baseline fit` algorithm. This is because OPT takes into account the impact on midgress while assigning traffic classes to clusters in the metro area. This significant improvement in midgress makes the case for implementing midgress-aware traffic provisioning algorithms in CDNs.

From Figure 4, we also see that `local search` performs quite well and gives a 15.44% reduction in midgress on average compared to `baseline fit`. `local search` also performs fairly well compared to OPT, with a modest 3.69% increase in midgress compared to OPT on average.

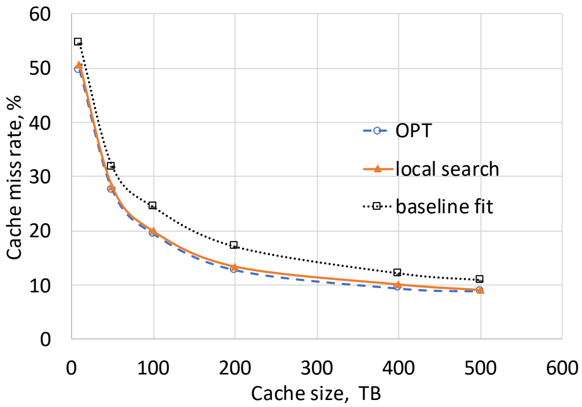


Figure 4: MRCs of OPT, `local search` and `baseline fit`.

4.1.1 How traffic provisioning impacts midgress

In Figure 5, we plot the cache miss rate of the 25 traffic classes when they are provisioned using OPT versus the midgress-unaware `baseline fit`, when the cumulative cache size of clusters in the metro area is 100TB. In addition, we also plot the average number of sites (across the 100 runs) that each traffic class is assigned to in Figure 6. From these figures, we see that OPT reduces the cache miss rate of 21 traffic classes when compared to `baseline fit`. In the case of traffic class 11, OPT results in almost 97% reduction in miss rate when compared to `baseline fit`. On the other hand, OPT increases the cache miss rate of four traffic classes, namely 4, 5, 13 and 19. By trading off the cache miss rates for these four traffic classes, OPT is able to reduce the overall midgress. But why does OPT choose this trade-off? There are three key insights that midgress-aware traffic provisioning takes into account to optimize midgress that `baseline fit` does not.

1) *In OPT's solution, traffic classes that have higher load, higher footprint pressure and greater MRC gradients get to occupy larger portions of the available cache space.* A traffic class has high footprint pressure if a large amount of unique content bytes is requested in a short period of time. This is true for traffic classes 11, 14, 15, 25 and 16 that account for 66.04% of the total load. OPT assigns traffic class 11 to two clusters because its load is greater than the capacity of a single cluster, resulting in that traffic class occupying 6 TB in one

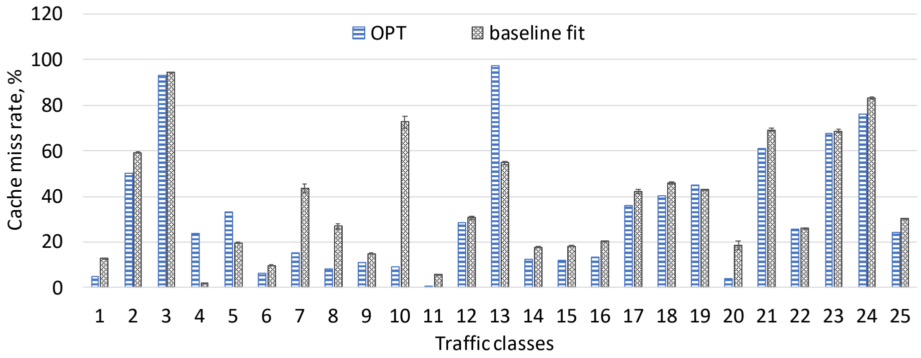


Figure 5: Average miss rate of each traffic class in a metro area of cache size 100 TB.

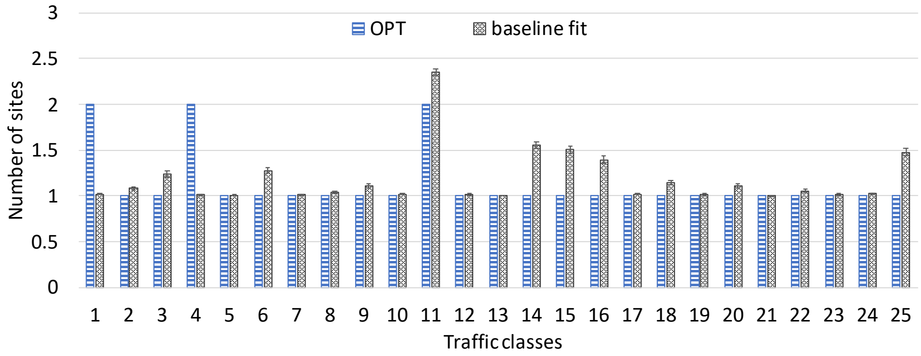


Figure 6: Average number of sites each traffic class is assigned to in a metro area of cache size 100 TB.

cluster with a miss rate of 0.5% and 7 TB in another cluster with a miss rate of nearly 0%. OPT also assigns an entire cluster each to traffic classes 14, 15, 25 and 16.

2) *OPT may split a traffic class and assigns it to multiple clusters if it has a relatively flat MRC.* This is true of traffic class 1 which has a relatively flat MRC and is assigned to two clusters. By reducing its footprint pressure in each of its assigned clusters, traffic class 1 is able to cede cache space to other traffic classes that are in more need.

3) *In OPT's solution, traffic classes that have lower footprint pressure occupy smaller portions of the available cache space.* This is true for traffic classes 4, 5 and 13. It also happens to be the case that these 3 traffic classes have very low load among the traffic classes considered. Both these factors render a higher cache miss rate relative to `baseline fit` that is midgress unaware. Note that low load alone does not indicate that it will occupy a smaller portion of the cache. For instance, traffic class 24 has moderate load but it has high footprint pressure and a greater MRC gradient, and ends up occupying 4.2 TB in one cluster.

4.2 Cluster-level traffic provisioning

The goal of cluster-level load balancing is to assign traffic classes to servers such that the midgress of the cluster is minimized. In our evaluation, we take the output of metro-level traffic provisioning from Section 4.1 that assigns traffic classes to each cluster and treat them as the inputs to cluster-level traffic provisioning. In this manner, we are able to understand the additional midgress reduction that is achievable by performing optimization at the cluster level, given that the metro level has already been optimized.

For cluster-level traffic provisioning, each traffic class defined at the metro-level is typically split into multiple finer-grained subclasses. The subclasses allow better allocation of traffic classes within a cluster. Traffic class 14 has very high load and hence was assigned to a cluster all by itself (Figure 6) by OPT at the metro-level. We considered that cluster for our evaluation of cluster-level traffic provisioning. Traffic class 14 consisted of 66 traffic subclasses that must be assigned to the 10 servers within a cluster, each server with a 1 TB cache.

OPT reduced the midgress for traffic class 14 by 31.26% after the metro-level optimization, when compared to the midgress achieved by `baseline fit`. Further, after using OPT for cluster-level provisioning, the midgress for traffic class 14 reduced further by 14.26%. In aggregate, the overall reduction of the midgress due to both provisioning steps of OPT is 41.07%, when compared to the baseline. Algorithm `local search` provided nearly as much reduction as OPT. For instance, `local search` provided a midgress reduction of 35.49%, compared to the baseline. However, `local search` was much faster and completed within 2 minutes, as opposed to the nearly 40,000 s (∼11 hours) taken by OPT.

4.3 Robustness to cache management policies

So far, we have developed traffic provisioning algorithms that model an LRU cache and evaluated the midgress reduction resulting when the sites also use LRU. The past decades have seen much academic research on numerous cache management algorithms that admit and evict objects using some combination of *recency* of access, *frequency* of access and object *size* to reduce cache miss rates (see Table 2 of [4]). We show that midgress-aware traffic provisioning algorithms proposed in this work, that model an LRU cache, achieve significant midgress reduction even when a CDN *does not* actually implement LRU at its sites.

We choose three typical algorithms from the literature for our evaluation. The first is an LRU variant called second-hit-LRU (or, SH-LRU) where the object is admitted to an LRU cache on second hit. The second is segmented LRU (SLRU) [40] that uses both recency and frequency for cache management. Finally, we implement the Greed-Dual-Size-Frequency (GDSF) [13] that uses all three of recency, frequency and size. Our evaluation uses the same cluster-level scenario as described in Section 4.2, where the goal is to assign the 66 traffic subclasses of traffic class 14 across 10 servers of size 1 TB each. First, we solve OPT that models LRU to get the optimal traffic class assignment across all servers within the cluster. The midgress of OPT's assignment is then computed by simulating the different cache management algorithms using the request traces of the subclasses. For comparison, we use the midgress-unaware `baseline fit` for traffic provisioning followed by a trace-based simulation of the different cache management algorithms to provide a baseline. When LRU cache management is used, OPT reduces the midgress by 13.3% when compared to `baseline fit`. In comparison, OPT reduces the midgress by 7.78%, 8.45% and 7.76% for SH-LRU, SLRU and GDSF respectively. The midgress reduction for the non-LRU algorithms is not as much as that for LRU. However, the midgress reductions for other algorithms are still quite robust and significant. The reason is that even when other factors are used for cache management decisions, most reasonable algorithms *still* use recency of access in a very significant way, and recency is well-captured in OPT. It is plausible that OPT can be reformulated to capture other cache management policies besides LRU and such an extension is a topic for future work.

5 Extending midgress-aware provisioning

We propose two extensions of midgress-aware traffic provisioning that address constraints in production settings. The first extension enforces a minimum number of sites that a traffic class must be assigned to and the second extension enforces a maximum cache miss rate per traffic class.

All results presented until now are for shared caches. While partitioned caches are not commonly used in production settings due to the overheads of dynamically resizing those partitions, there is increasing interest to implement and evaluate partitioned caches [1,5,9,14,16,21,25,36,43,44,46,62,67,69]. We propose a third extension to show that our traffic provisioning approach can reduce midgress in partitioned caches.

5.1 Minimum redundancy guarantee

Let M_j be the minimum number of sites that traffic class j should be assigned to. M_j is an integer $\in [1,M]$, where M is the total number of sites. Let y_{ij} be an indicator variable that is set to 1 when $x_{ij} > 0$ and 0 otherwise. Then, the load assignment constraint in Section 2.2 is appended to include the following minimum redundancy constraints.

$$y_{ij} = \lceil x_{ij} \rceil \quad \forall\, j = 1 \ldots N \tag{9}$$

$$\sum_{i=1}^{M} y_{ij} >= M_j \quad \forall\, j = 1 \ldots N \tag{10}$$

$$y_{ij} \in \{0,1\} \tag{11}$$

Equations 9 and 10 ensure that traffic class j is assigned to at least M_j sites. We call the modified optimization model OPT-M. The additional constraints are affine and they do not increase the complexity of the optimization problem. Both `local search` and `baseline fit` can also be modified to incorporate the redundancy constraint by simply ensuring that each traffic class j is assigned to at least M_j sites in each (re-)provisioning step.

Experimental evaluation. We measure the reduction in midgress by OPT-M and the modified `local search` when compared to the modified `baseline fit`. We use the same evaluation parameters as Section 4.1 where the cache size of each cluster in the metro area is 10 TB. We find that the cache miss rates increase with increase in redundancy for all three algorithms. We also find that the cache miss rate of `baseline fit` with minimum redundancy = 1 (resp. 2) is similar to the cache miss rate of `local search` and OPT-M with minimum redundancy 2 (resp. 3). This shows that midgress-aware traffic provisioning can provide the same midgress as `baseline fit` with added redundancy.

5.2 Maximum cache miss rate guarantee

Let MR_j be the maximum cache miss rate for traffic class j. Then, the optimization model in Section 2.2 can be extended to incorporate the maximum cache miss rate guarantee.

$$\sum_{i=1}^{M} x_{ij} m_j(c_{ij}) \leq MR_j \quad \forall j = 1 \ldots N \tag{12}$$

Equation 12 states that the average miss rate of traffic class j across all M sites should be at most MR_j. We call the modified optimization model, OPT-MR. Equation 12 is a biconvex constraint and doesn't increase the complexity of the problem.

We make two modifications to `local search`. First, `baseline fit` in the first step does not always provide a feasible solution when $MR_j < 100\%$. This is because `baseline fit` is midgress unaware. Hence, we start with all traffic classes being unassigned. Second, the re-provisioning step assigns a traffic class to a site only when the miss rate guarantees of all traffic classes assigned to that site are not violated.

Infeasible solutions. OPT-MR can be infeasible in cases where certain traffic classes fail to meet the maximum cache miss rate MR_j guarantee. For example consider a cache size of 10 TB. The lowest miss rate that traffic class 3 (Table 3) can possibly achieve at 10 TB is 91%. Hence, any maximum cache miss rate target less than 91% cannot be achieved.

Experimental evaluation. We choose traffic classes 13, 23 and 24 that have high miss rates in OPT and set their maximum cache miss rates to 70%. We evaluate the performance of metro-level traffic provisioning under the same conditions as in Section 4.1 where the cache size is 10 TB.

OPT-MR returns a feasible solution. The overall miss rate of the metro area is 20.04%, a 3.24% increase in midgress compared to OPT. In the process, three traffic classes experience a significant increase in their respective miss rates relative to OPT. The miss rate of traffic class 2 increases from 50.12% to 65.85%, of traffic class 17 from 36.11% to 54.2% and of traffic class 21 from 61.22% to 68.01%. This is because traffic classes 13 and 24 occupy more cache space with OPT-MR than they do in OPT, so they meet their miss rate guarantee, despite traffic class 13 having the lowest load and low footprint pressure, and traffic class 24 having low load.

We run the modified `local search` 100 times with different random orderings of the input traffic classes. `local search` returns a feasible solution 67% of the time indicating that its feasibility depends on the ordering of the traffic class inputs. For feasible assignments, `local search` has an average miss rate of 21.69%, about 8.23% more than that of OPT-MR. `baseline fit` is footprint-unaware and cannot guarantee cache miss rates.

5.3 Traffic provisioning in partitioned caches

In a partitioned cache, each traffic class is assigned to its own separate cache partition and each partition performs evictions independently. Production CDNs do not typically implement partitioned caches due to the significant overheads involved in implementing and dynamically maintaining the partitions. However, we show that our optimization model for midgress-aware traffic provisioning can be extended to work with partitioned caches.

Modeling and implementing traffic provisioning for partitioned caches. In partitioned caches, every traffic class occupies a separate partition with its own LRU list, assuming the LRU eviction policy. Thus, the eviction age of each traffic class assigned to the same cache can be different. Therefore, the optimization model for midgress-aware traffic provisioning for partitioned caches is the same as that of OPT (Section 2.2), minus the eviction age constraint. Hence, Equations 1-3 and 5-8 accurately model midgress minimization for partitioned caches. We call this modified model OPT-part.

We implement a baseline midgress-unaware algorithm called `baseline fit-part` for partitioned caches that is based on consistent hashing. The algorithm takes as input the set of N traffic classes and the set of M sites that are both hashed to points on a unit circle. Each traffic class j is assigned to the nearest site i on the unit circle in the clockwise direction. If the chosen site i does not have enough capacity to host the entire load λ_j, then a first fit algorithm is used, starting from the chosen site i, and continuing to subsequent sites on the circle in the clockwise direction, until all load is assigned. After all traffic class assignments are made, in each site, we determine the sizes of the partitions that host each traffic class. To compute the partition sizes, we use a known gradient descent algorithm [62] that minimizes the midgress of each site. The total midgress achieved by `baseline fit-part` is then the sum of the midgress across all sites.

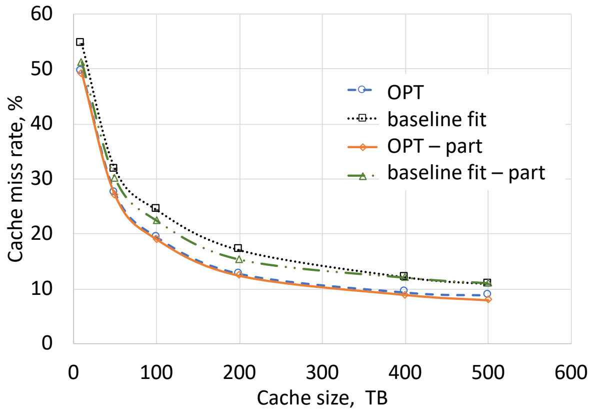


Figure 7: MRC of OPT and baseline fit on shared and partitioned caches.

Experimental evaluation. We evaluate `baseline fit-part` and OPT-part at the metro-level using production traces described in Section 4 and at different cache sizes per cluster of 1 TB, 5 TB, 10 TB, 20 TB, 40 TB and 50 TB. We report the average cache miss rate of 100 runs. The 95% confidence intervals of the expected cache miss rates have a margin of error of less than 0.4%.

As shown in Figure 7 we find that OPT-part reduces midgress when compared to `baseline fit-part` by more than 14%, on average across the different cache sizes. Thus midgress-aware traffic provisioning can significantly reduce the midgress even for partitioned caches. As comparison, in Figure 7, we also plot OPT and `baseline fit` that we described for shared caches in Sections 2 and 3. Interestingly, we find that the cache miss rate of OPT-part is only 0.49% less than that of OPT, on average across the different cache sizes. Hence, while OPT-part has the lowest cache miss rate, OPT has nearly the same miss rate without the additional overhead of cache partitioning.

5.3.1 Implementing cache partitioning for traffic provisioning in production settings

In the previous section, we have seen that cache partitioning can further reduce the midgress of a metro area since each traffic class occupies a separate partition and traffic classes assigned to the same site could have different eviction ages. But it is not implemented in practice for traffic provisioning due to the following reasons.

1) In large CDNs, many different traffic classes must share a cache. Further, the mix of traffic classes sharing a cache and their respective loads change frequently over time at the whim of the load balancer. To obtain the benefits of cache partitioning, the partitions need to be constantly resized by shrinking cache space for certain traffic classes and expanding the cache space for others. Such resizing is resource intensive. Further, constantly resizing dynamic partitions may not lead to lower midgress, especially during transitions between cache sizes. While partitioning the cache statically is easier to implement, static partitions do not adjust to changes in the traffic mix, leading to sub-optimal performance. On the other hand, an unpartitioned shared cache dynamically adjusts the cache space occupied by different traffic classes based on the load and the traffic characteristics, without the need for complex (re)partitioning operations.

2) From the previous section, we see that traffic provisioning in a shared unpartitioned cache provides nearly the same midgress as a partitioned cache. Thus, there is little incentive to redesign the traffic provisioning system to work with partitioned caches and incurring the additional software complexity and resource overhead.

For the reasons outlined above, the shared unpartitioned cache studied in our work is the implementation of choice for many major CDNs, including Akamai.

6 Related work

Traffic provisioning in CDNs has been studied in the context of load balancing. However, the load balancing algorithms focus on ensuring that servers are not overloaded and do not explicitly minimize midgress. Likewise, minimizing cache misses through better cache management policies has a rich literature. However, we view cache management as complementary to midgress-aware traffic provisioning. We review relevant existing literature in these areas.

Load balancing. Request redirection schemes at the network layer, based on DNS [8,32], and at the application layer, based on URL rewriting or HTTP redirection [48], have been proposed to load balance traffic across multiple servers. Dynamic load balancing algorithms [10, 11, 71] continuously measure the load on different servers and load balance end-user requests to improve performance. Consistent-hashing and randomized load balancing algorithms [41, 42, 54, 55] have also been proposed to load balance end-user requests in content delivery systems. Extensions to traditional load balancing, that minimize the energy consumption of CDNs [50] have also been proposed in the literature. Much of the work above are in the context of routing user requests in real-time to servers. But, they can be adapted to our context of performing (offline) traffic provisioning, a step that precedes request routing in a production CDN. *However, there is no prior work on explicitly minimizing midgress.*

Cache management. There has been a significant amount of research on cache management policies to minimize cache miss rates [2, 3, 6, 17–19, 22, 26–31, 33, 37, 38, 45, 49, 52, 57, 58, 60, 64, 68, 70]. Some proposed caching policies include Adapt-Size [4], Cliffhanger [15], SLRU [40], TLRU [23], S4LRU [34], CFLRU [59], ARC [53], LRU-S [65], LRU-K [56], and GDS [7]. Dynamically partitioning the cache to reduce miss rates has also been explored [1, 5, 9, 14, 16, 21, 25, 36, 43, 44, 46, 62, 67, 69]. However, production CDNs do not employ dynamically-partitioned caches since it introduces significant performance and operational overheads. We view work on cache management as a complementary technique to traffic provisioning, both with the goal of midgress reduction.

Recent work on footprint descriptors [66] is focused on efficient techniques for evaluating the miss rates of traffic mixes. We use footprint descriptors to quickly compute the midgress of a traffic class assignment, as well as to efficiently compute the MRC and eviction age function of traffic classes. However, the work on footprint descriptors does not minimize midgress.

7 Conclusion

We propose midgress-aware traffic provisioning that explicitly minimizes the midgress traffic of a CDN, while ensuring that no server or cluster is overloaded. Using extensive traces for 25 traffic classes from Akamai's CDN, we show that the midgress of a metro can be reduced by 18.37% when compared to a midgress-unaware baseline. We propose a midgress-aware heuristic, `local search`, that provisions traffic classes to achieve a midgress reduction that is within 1.1% of the optimum, and is very fast and well suited for production settings. We also show that using our traffic provisioning algorithms at the cluster level results in significant reductions in midgress. Given that a large CDN can have midgress of over 10 Tbps, even a small reduction in midgress can result in millions of dollars of savings per year. Our work provides a strong case for implementing midgress-aware provisioning in CDNs.

8 Acknowledgments

We thank our reviewers and our shepherd Kiran Kumar Muniswamy Reddy for their great feedback. This research was supported in part by NSF grant CNS-1763617.

References

[1] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *FAST*, pages 355–369, 2016.

[2] Daniel S. Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Perform. Eval.*, 79:2 – 23, 2014. Special Issue: Performance 2014.

[3] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.

[4] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *NSDI*, pages 483–498, 2017.

[5] Sem Borst, Varun Gupta, and Anwar Walid. Distributed caching algorithms for content distribution networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. Citeseer, 2010.

[6] PJ Burville and JFC Kingman. On a model for storage and search. *Journal of Applied Probability*, pages 697–701, 1973.

[7] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems*, volume 12, pages 193–206, 1997.

[8] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Request redirection algorithms for distributed web systems. *IEEE transactions on parallel and distributed systems*, 14(4):355–368, 2003.

[9] Damiano Carra and Pietro Michiardi. Memory partitioning and management in memcached. *IEEE Transactions on Services Computing*, 2016.

[10] Robert L Carter and Mark E Crovella. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 3, pages 1014–1021. IEEE, 1997.

[11] Chung-Min Chen, Yibei Ling, Marcus Pang, Wai Chen, Shengwei Cai, Yoshihisa Suwa, and Onur Altintas. Scalable request routing with next-neighbor load sharing in multi-server environments. In *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, volume 1, pages 441–446. IEEE, 2005.

[12] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.

[13] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[14] Weibo Chu, Mostafa Dehghan, John CS Lui, Don Towsley, and Zhi-Li Zhang. Joint cache resource allocation and request routing for in-network caching services. *Computer Networks*, 131:1–14, 2018.

[15] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, pages 379–392, 2016.

[16] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *Usenix ATC*, 2017.

[17] Edward G. Coffman and Predrag Jelenković. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Operations Research Letters*, 25:109–118, 1999.

[18] Edward Grady Coffman and Peter J Denning. *Operating systems theory*. Prentice-Hall, 1973.

[19] Asit Dan and Don Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, 1990.

[20] Mostafa Dehghan, Weibo Chu, Philippe Nain, Don Towsley, and Zhi-Li Zhang. Sharing cache resources among content providers: A utility-based approach. *IEEE/ACM Transactions on Networking (TON)*, 27(2):477–490, 2019.

[21] Mostafa Dehghan, Laurent Massoulie, Don Towsley, Daniel Menasche, and Yong Chiang Tay. A utility optimization approach to network cache design. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[22] Robert P Dobrow and James Allen Fill. The move-to-front rule for self-organizing lists with Markov dependent requests. In *Discrete Probability and Algorithms*, pages 57–80. Springer, 1995.

[23] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *IEE Euromicro PDP*, pages 146–153, 2014.

[24] Ronald Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[25] Michal Feldman and John Chuang. Service differentiation in web caching and content distribution. In *Proceedings of the IASTED International Conference on Communications and Computer Networks*, 2002.

[26] James Allen Fill and Lars Holst. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms*, 8:179–186, 1996.

[27] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39:207–229, 1992.

[28] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, page 8, 2012.

[29] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE*, pages 395–396, 2012.

[30] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS*, pages 123–136, 2015.

[31] Erol Gelenbe. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers*, 100:611–618, 1973.

[32] Michel Goemans. Load balancing in content deliverynetworks. *MA Annual Program Year Workshop:Network Management and Design*, April 2003.

[33] WJ Hendricks. The stationary distribution of an interesting Markov chain. *Journal of Applied Probability*, pages 231–233, 1972.

[34] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[35] Cisco Visual Networking Index. The zettabyte era: Trends and analysis. June 2017.

[36] Stratis Ioannidis and Edmund Yeh. Jointly optimal routing and caching for arbitrary network topologies. *IEEE Journal on Selected Areas in Communications*, 2018.

[37] Predrag R Jelenković. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability*, 9:430–464, 1999.

[38] Predrag R Jelenković and Ana Radovanović. Least-recently-used caching with dependent requests. *Theoretical computer science*, 326:293–327, 2004.

[39] Poul-Henning Kamp. Varnish LRU architecture, June 2007. Available at `https://www.varnish-cache.org/trac/wiki/ArchitectureLRU`, accessed 09/12/16.

[40] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, (3):38–46, 1994.

[41] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[42] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[43] Terence Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey MacKie-Mason. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value. 1999.

[44] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[45] W. Frank King. Analysis of demand paging algorithms. In *IFIP Congress (1)*, pages 485–490, 1971.

[46] Bong-Jun Ko, Kang-Won Lee, Khalil Amiri, and Seraphin Calo. Scalable service differentiation in a shared storage cache. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 184–193. IEEE, 2003.

[47] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[48] Sabato Manfredi, Francesco Oliviero, and Simon Pietro Romano. A distributed control law for load balancing

in content delivery networks. *IEEE/ACM Transactions on Networking (TON)*, 21(1):55–68, 2013.

[49] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014.

[50] Vimal Mathew, Ramesh K Sitaraman, and Prashant Shenoy. Energy-aware load balancing in content delivery networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 954–962. IEEE, 2012.

[51] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[52] John McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965.

[53] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST*, volume 3, pages 115–130, 2003.

[54] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. *arXiv preprint arXiv:1608.01350*, 2016.

[55] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[56] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD*, 22(2):297–306, 1993.

[57] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. An optimality proof of the LRU-K page replacement algorithm. *JACM*, 46:92–112, 1999.

[58] Antonis Panagakis, Athanasios Vaios, and Ioannis Stavrakakis. Approximate analysis of LRU in the case of short term correlations. *Computer Networks*, 52:1142–1152, 2008.

[59] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES*, pages 234–241, 2006.

[60] Konstantinos Psounis, An Zhu, Balaji Prabhakar, and Rajeev Motwani. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks*, 45:379–398, 2004.

[61] Guocong Quan, Jian Tan, Atilla Eryilmaz, and Ness Shroff. A new flexible multi-flow lru cache management paradigm for minimizing misses. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):39, 2019.

[62] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 423–432. IEEE, 2006.

[63] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[64] Eliane R Rodrigues. The performance of the move-to-front scheme under some particular forms of Markov requests. *Journal of applied probability*, pages 1089–1102, 1995.

[65] David Starobinski and David Tse. Probabilistic methods for web caching. *Perform. Eval.*, 46:125–137, 2001.

[66] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67. ACM, 2017.

[67] Dominique Thiébaut, Harold S. Stone, and Joel L Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[68] Naoki Tsukada, Ryo Hirade, and Naoto Miyoshi. Fluid limit analysis of FIFO and RR caching for independent reference model. *Perform. Eval.*, 69:403–412, September 2012.

[69] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 381–392. IEEE, 2014.

[70] Neal E Young. Online paging against adversarially biased random inputs. *Journal of Algorithms*, 37:218–235, 2000.

[71] Zeng Zeng and Bharadwaj Veeravalli. Design and performance evaluation of queue-and-rate-adjustment dynamic load balancing policies for distributed networks. *IEEE Transactions on Computers*, 55(11):1410–1422, 2006.

GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks

Rui Wang[1], Yongkun Li[1], Hong Xie[2], Yinlong Xu[1], John C.S. Lui[3]
[1]*University of Science and Technology of China*
[2]*Chongqing University* [3]*The Chinese University of Hong Kong*

Abstract

Traditional graph systems mainly use the *iteration-based model* which iteratively loads graph blocks into memory for analysis so as to reduce random I/Os. However, this iteration-based model limits the efficiency and scalability of running random walk, which is a fundamental technique to analyze large graphs. In this paper, we propose GraphWalker, an I/O-efficient graph system for random walks by deploying a novel *state-aware I/O model* with *asynchronous walk updating*. GraphWalker is efficient to handle very large disk-resident graphs consisting of hundreds of billions of edges with only a single commodity machine, and it is also scalable to run tens of billions of random walks with thousands of steps long. Experiments on our prototype system show that GraphWalker can achieve more than an order of magnitude speedup when running a large amount of long random walks when compared with DrunkardMob, which is tailored for random walk based on the classical system GraphChi, as well as two state-of-the-art single-machine graph systems, Graphene and GraFSoft. Furthermore, comparing with the most recent distributed system KnightKing, which optimizes for random walks and runs on cluster machines, GraphWalker achieves comparable performance with only a single machine, thereby making it a more cost-effective alternative.

1 Introduction

To improve the performance of analyzing large graphs on a single-machine, many out-of-core graph processing systems are proposed [6, 10, 11, 20, 24, 29, 31, 37, 42, 43, 49]. One major effort of these systems is to reduce random disk I/Os. Generally, when a graph is too large to fit into the memory, these systems partition the entire graph into many subgraphs, and store each subgraph as a block on disk, e.g., *shard* in GraphChi [24]. To carry graph analysis, they adopt an *iteration-based model*. In each iteration, blocks are sequentially loaded into memory, then analysis related to the loaded subgraph is performed. This way, it turns massive random I/Os into a series of sequential I/Os, and guarantees synchronized analysis over all blocks in each iteration.

Random walks have been proven to be efficient to analyze large graphs [7,12,15,19,23,26,27,36,38]. For example, Personalized PageRank (PPR) [12, 23] starts thousands of walks from the source vertex to compute visit frequencies in order to approximate PageRank values. SimRank (SR) [19] computes the similarity for a vertex pair by first starting many random walks from each of the vertex pair, and then computing the expected meeting time. Random walk domination (RWD) [27] starts walks from all vertices to measure the influence diffusion over the whole graph. To compute PPR for all vertices, and all-pair similarity, it is also required to start random walks from every vertex, which results in massive concurrent walks.

We observe that current graph systems with the iteration-based model cannot efficiently support random walks. The major limitations are three folds. First, due to the high randomness nature, many walks are unevenly scattered at different parts of the graph, so some subgraphs may contain only few walks. However, the iteration-based model is unaware of these walk states, and just sequentially loads all needed subgraphs into memory for analysis, so it results in very low I/O utilization. Second, as the iteration-based model ensures a synchronized analysis, all walks move exactly one step in each iteration. As a result, the walk updating efficiency is also limited and thus further exacerbates the I/O efficiency. This is true especially for applications demanding long walks. Lastly, due to the randomness of walks, the number of walks at each vertex varies dynamically, so existing graph systems usually use massive dynamic arrays to record the walks currently traveling through each edge or each vertex in the graph. However, this indexing design requires large memory space and thus limits the scalability of handling very large graphs.

Various design efforts are made in recent years to improve the I/O efficiency of the iteration-based model, e.g., DynamicShards [43] and Graphene [29] dynamically adjust the layout of graph blocks to reduce the loading of useless data in each iteration. CLIP [6] proposes the *re-entry scheme* and Lumos [42] proposes the *cross-iteration value propagation technique*, and both of them aim to make full use of the loaded blocks to avoid loading the corresponding graph portions in future iterations. These systems greatly improve the performance, but they do not take into account the random walk features. To efficiently support parallel random walks, DrunkardMob [23]proposes several optimizations to reduce the memory usage of walk indexes so as to support a large amount of random walks. However, its scalability is still limited, e.g., it costs 2.3 hours to run one billion random walks with ten-step long on a medium-scale graph

YahooWeb [5], and it is even unable to run random walks on very large graphs like CrawlWeb [3] due to its high memory consumption. KnightKing [46] is the most recent distributed graph system which is also optimized for random walks. It provides a unified framework to support various random walks, and mainly focuses on optimizing the walking process without addressing disk I/Os.

To address the I/O efficiency problem so as to efficiently support fast and scalable random walks, we develop GraphWalker, which is an I/O-efficient and resource-friendly graph system. GraphWalker mainly focuses on improving the I/O efficiency by developing a state-aware I/O model with asynchronous walk updating. It also utilizes a lightweight block-centric walk management scheme to improve memory efficiency. In summary, our main contributions are as follows.

- We develop a novel *state-aware I/O model*, which leverages the state of each random walk to preferentially load the graph block with the most walks from disk into memory, so as to improve the I/O utilization. We also propose a walk-conscious caching scheme to improve cache efficiency.
- We adopt an *asynchronous walk updating scheme* based on the *re-entry* method [6], which allows each walk to move as many steps as possible so as to fully utilize the loaded subgraph and greatly accelerate the progress of random walks. To address the straggler issues caused by asynchronous update, we also employ a probabilistic approach to balance the progress of each walk.
- We propose a *lightweight block-centric indexing scheme* to manage walk states and adopt a fixed-length walk buffering strategy to reduce the memory cost for recording walk states. We also develop a disk-based walk management scheme and use asynchronous batched I/Os to write walk states back to disk so as to support running massive random walks in parallel on huge graphs.
- We implement a prototype and conduct extensive experiments to demonstrate its efficiency. Results show that GraphWalker can achieve more than *an order of magnitude speedup* compared with the random-walk-specific system DrunkardMob [23], as well as two state-of-the-art single-machine graph systems, Graphene [29] and GraFSoft [20]. Furthermore, GraphWalker is more resource friendly as its performance is even comparable with the state-of-the-art distributed random walk system KnightKing [46] running on a cluster of machines.

2 Background and Motivation

We first introduce the storage and computation process of the iteration-based model, then analyze its limitations in supporting random walks.

2.1 Iteration-based Graph Computation

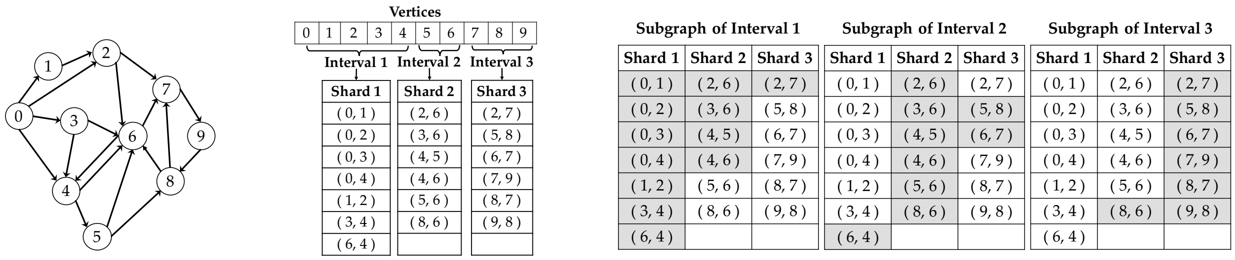
For simplicity, we take GraphChi [24], the pioneering single-machine iteration-based graph system, as an example to illustrate its key idea. We like to point out that this iteration-based model is widely used in many graph systems like [10, 11, 20, 29, 37, 42, 43, 49]. GraphChi splits all vertices into disjoint intervals and associates each interval with a *shard*, which stores all the edges whose destination vertices lie in this interval. Edges in each shard are sorted according to their source vertices. For example, for the graph in Figure 1(a), its data organization in shards is illustrated in Figure 1(b).

To perform analysis, GraphChi loads all subgraphs iteratively by using the parallel sliding window (PSW), which is illustrated in Figure 1(c). In each iteration, it loads the subgraphs in a round-robin order and guarantees synchronization between all computation tasks over the whole graph. Specifically, at each time slot, GraphChi loads one subgraph corresponding to one interval into memory for analysis. It first loads the in-edges from its corresponding shard, then loads the out-edges from other shards. As edges are sorted by source vertices in each shard, at most P sequential disk reads are needed to load the subgraph corresponding to one interval if there are P shards. Then GraphChi traverses the vertices of the loaded subgraph and conduct computation. This way, GraphChi transfers random accesses to a series of sequential accesses and greatly improves the performance of disk-resident graph processing.

2.2 Limitations in Supporting Random Walks

A random walk proceeds by starting at a source vertex, then repeats the process of randomly selecting a neighbor to visit. Many applications often need to simultaneously run massive random walks [12, 27, 44, 47]. When supporting massive parallel random walks, graph systems with the iteration-based model suffer from several limitations, e.g., low I/O utilization and low walk updating rate, as well as high memory cost for managing walks. In the following, we analyze these limitations in details.

Limitation 1: Low I/O utilization. First of all, the iteration-based model leads to low I/O utilization for random walks, which is defined as the number of edges used for updating walks divided by the number of edges loaded in one I/O, i.e., a subgraph loading. The main reason is that walks may be unevenly scattered across the entire graph after a few steps even if they started from the same source vertex. As a result, even if there are only few walks in some blocks, they are still required to be loaded into memory, so it brings extremely low I/O utilization. Some recent works like DynamicShards [43] and Graphene [29] adopt an *on-demand* I/O strategy to dynamically adjust graph block layout and skip loading blocks which do not contain any walks so as to reduce the loading of useless edges, but the low I/O utilization problem is still not fully addressed. As long as there is one walk in a block, then this block still has to be loaded into memory for computation.



Shard 1	Shard 2	Shard 3
(0, 1)	(2, 6)	(2, 7)
(0, 2)	(3, 6)	(5, 8)
(0, 3)	(4, 5)	(6, 7)
(0, 4)	(4, 6)	(7, 9)
(1, 2)	(5, 6)	(8, 7)
(3, 4)	(8, 6)	(9, 8)
(6, 4)		

Subgraph of Interval 1			Subgraph of Interval 2			Subgraph of Interval 3		
Shard 1	Shard 2	Shard 3	Shard 1	Shard 2	Shard 3	Shard 1	Shard 2	Shard 3
(0, 1)	(2, 6)	(2, 7)	(0, 1)	(2, 6)	(2, 7)	(0, 1)	(2, 6)	(2, 7)
(0, 2)	(3, 6)	(5, 8)	(0, 2)	(3, 6)	(5, 8)	(0, 2)	(3, 6)	(5, 8)
(0, 3)	(4, 5)	(6, 7)	(0, 3)	(4, 5)	(6, 7)	(0, 3)	(4, 5)	(6, 7)
(0, 4)	(4, 6)	(7, 9)	(0, 4)	(4, 6)	(7, 9)	(0, 4)	(4, 6)	(7, 9)
(1, 2)	(5, 6)	(8, 7)	(1, 2)	(5, 6)	(8, 7)	(1, 2)	(5, 6)	(8, 7)
(3, 4)	(8, 6)	(9, 8)	(3, 4)	(8, 6)	(9, 8)	(3, 4)	(8, 6)	(9, 8)
(6, 4)			(6, 4)			(6, 4)		

(a) Example graph (b) Data organization in shards (c) Graph loading with parallel sliding window

Figure 1: Storage and I/O model in GraphChi

We also run experiments to demonstrate the skewed walk distribution and low I/O utilization. We use DrunkardMob to run 10^4, 10^6 and 10^8 walks of length ten on the Friendster graph consisting of 68.3 million vertices, and consider starting random walks from a single source (SSRW) or multiple random sources (MSRW). Please refer to §4.1 for detailed experiment setting. Figure 2(a) shows the average I/O utilization, which is 3.1×10^{-6}, 3.2×10^{-4} and 0.032 for SSRW with 10^4, 10^6 and 10^8 walks, respectively, and the results are similar for MSRW. We point out that the I/O utilization is very low, especially for the case of small number of walks. Figure 2(b) further shows the distribution of walks over blocks after four iterations when running 10^6 walks started from a single source, which demonstrates heavily skewed distribution. We also run the same experiments with Graphene, the average I/O utilization is 6.1×10^{-3}, 3.1×10^{-3} and 0.032 for MSRW when running 10^4, 10^6 and 10^8 walks, respectively. We find that Graphene can greatly improve the I/O utilization when the number of walks is small, but the I/O efficiency is still limited when running massive walks.

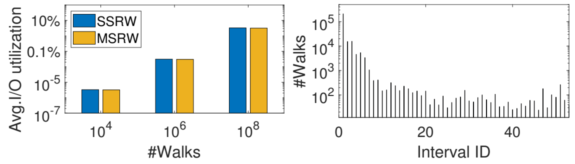
In our GraphWalker, we propose a *state-aware I/O model*, which loads graph blocks by considering the states of walks. Precisely, it always preferentially chooses to load the block with the maximum number of walks so as to make more walks get updated by using an I/O. Our experiment results show that GraphWalker brings 2× to 4× I/O utilization (see §4.2.3).

Limitation 2: Low walk updating rate. The iteration-based model also leads to low walk updating rate, which is defined as the sum of walked steps of all walks in the loaded subgraph divided by the total steps needed to walk. This is because with the iteration-based model, each walk can only move one step in each iteration in a synchronized pace, which severely wastes the data in memory as many walks can still make more moves over the loaded subgraph. To demonstrate, we run 10^6 random walks started from a single vertex by using the same setting as above. Figure 3(a) shows the walk updating rate. We find that all walks together move only 1K steps on average in one I/O, except for the first one, but we have total 10^9 steps to walk, so the updating rate is as low as 10^{-6}. We also count the fraction of walks that still remain in the first block in each iteration as shown in Figure 3(b). We find that on average, 75.3% walks still remain in the first block, and they could move more steps in the current iteration, so it results in the low walk updating rate. Recently, CLIP [6] proposes a *re-entry* method and Lumos [42] proposed the *cross-iteration value propagation* technique to reuse the loaded data to improve the I/O and computing efficiency, but it also brings extra cost as it accesses the whole subgraph multiple times.

In GraphWalker, we propose an *asynchronous walk updating scheme* based on the *re-entry* technique to allow walks to move as many steps as possible within the currently loaded subgraph without extra subgraph accesses.With our asynchronous walk updating scheme, GraphWalker greatly increases the walk updating rate and reduces the completion time of all walks. We also develop a probabilistic approach to balance walk progress so as to address the straggler issues.

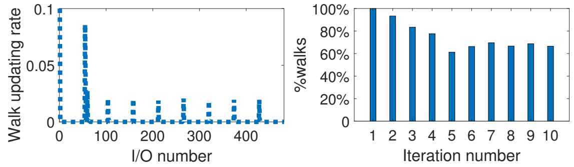
Limitation 3: High memory cost for managing walk data. Since the number of walks at each vertex is dynamic and unpredictable, walks are usually stored with massive dynamic arrays, e.g., GraphChi associates each edge with a dynamic array to store the walks currently traveling through the edge. This design incurs high memory cost, e.g., it needs at least 26.4 GB space to store only the walk array indexes, not including the walk states information, for a medium scale graph like YahooWeb [5], which has 1.4 billion vertices and 6.6 billion edges.Some systems use a vertex-centric way to manage walks [6, 10, 29], but it also incurs high memory cost, e.g., 5.6 GB to store the walk array indexes for YahooWeb.

DrunkardMob encodes the states of a walk into a 32-bit or 64-bit representation and puts walks of adjacent 128 vertices



(a) I/O utilization (b) Distribution of walks

Figure 2: I/O utilization under different walk settings and the distribution of number of walks over blocks (intervals).



(a) Walk updating rate (b) Walks in the 1^{st} interval

Figure 3: Walk updating rate and the fraction of walks that still remain in the first block in each iteration.

into the same walk buffer to reduce the total size of walk indexes. It reduces the size of walk array indexes to 1/128 of that of the vertex-centric management, e.g., only 44.8 MB for YahooWeb. However, each walk buffer in DrunkardMob is also managed with a dynamic array, so it still suffers from the scalability problem. First, as it creates too many dynamic arrays for large graphs, e.g., 11.2 million for YahooWeb, it causes frequent memory re-allocation, which not only introduces memory fragmentation, but also brings extra time cost and limits the graph scale that could be analyzed. Second, DrunkardMob keeps all walks in memory, so the number of walks is limited by memory space, e.g., 10 billion walks cost at least 40GB memory. Besides, it also incurs high cost to flush walk indexes to disk as they are related to many files.

In our GraphWalker, we adopt a block-centric method to manage walk data, so it greatly reduces the size of walk indexes. We also use fixed-length buffers to cache walks so as to avoid frequent memory re-allocation. With our lightweight scheme, both the scale of graphs and the number of walks that can be handled are no longer limited by memory capacity.

3 Design of GraphWalker

In this section, we first introduce the main idea of GraphWalker, which is an I/O-efficient and resource-friendly design targeted for random walks on single machine. We then present the details of its key design techniques, including state-aware graph loading, asynchronous walk updating, and lightweight walk management.

3.1 Main Idea

We target for supporting not only a very large number of walks, say tens of billions of walks, but also very long walks, say thousands of steps for each walk. To achieve this goal, the main idea is to adopt a *state-aware model* which leverages the states of each walk, e.g., the current vertex at which the walk stays. Briefly speaking, unlike the iteration-based model which blindly loads graph blocks sequentially, the state-aware model chooses to load the graph block containing the largest number of walks, and makes each walk move as many steps as possible until it reaches the boundary of the loaded subgraph. By doing this, walks can get updated as much as possible within each I/O. As a result, both the low I/O utilization and low walk updating rate problems can be efficiently addressed.

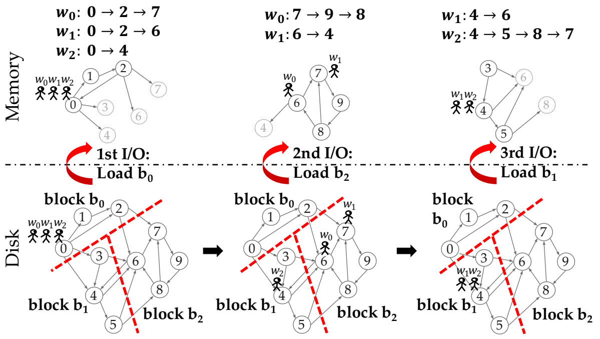


Figure 4: Main idea of the state-aware model

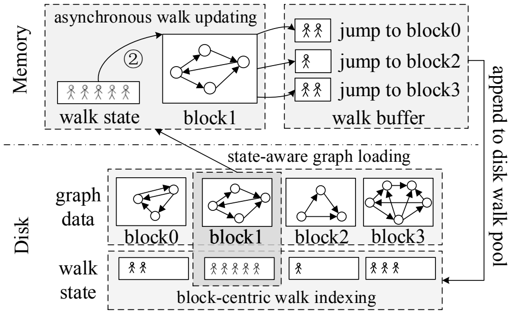


Figure 5: Overall design of GraphWalker

To further illustrate the above idea and analyze its benefits, we still consider the example graph in Figure 1(a). Suppose that we have to run three random walks which start at node 0 and have to move four steps. Figure 4 shows the process of graph loading and walk updating with the state-aware model. Specifically, in the first I/O, graph block b_0 is loaded into memory as it contains all the three walks. With the loaded graph block b_0, walk w_0 and w_1 move two steps, and w_2 moves only one step as it requires other graph blocks which are not in memory for walking more steps. As two walks fall into block b_2, in the second I/O, block b_2 is loaded into memory, and walk w_0 finishes and w_1 can move one step. Finally, both the remaining two walks are in block b_1, so we load b_1 into memory, and all walks can be finished. Note that only three I/Os are required in this example. However, for the iteration-based model, it may need 12 I/Os, because it uses four iterations, and generates three I/Os in each iteration.

Remark. We would like to emphasize that the state-aware model is different from the on-demand I/O model proposed in DynamicShards [43] and Graphene [29]. Note that the on-demand I/O model dynamically adjusts the graph blocks layout in each iteration and skips the blocks without containing any walks, but it still follows the iteration-based manner. Besides, even if there is only one walk in a block, it has to load the block into memory for analysis.

Based on the above idea, we develop an I/O-efficient graph system, GraphWalker, which supports fast and scalable random walks. GraphWalker mainly consists of three parts: (1) *State-aware graph loading,* (2) *Asynchronous walk updating,* and (3) *Block-centric walk management*. The overall design of GraphWalker is also illustrated in Figure 5. In the following subsections, we present its design in details.

3.2 State-Aware Graph Loading

Graph data organization and partition. GraphWalker manages graph data with the widely used *Compressed Sparse Row (CSR)* format, which sequentially stores the out neighbors of vertices as a *csr file* on disk, and uses an *index file* to record the beginning position of each vertex in the *csr file*. GraphWalker partitions a graph into blocks according to vertex IDs. Specifically, we sequentially add vertices and

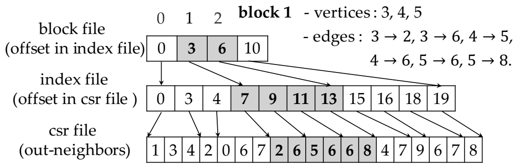


Figure 6: Graph data organization of the example graph in Figure 1(a). The graph is stored in CSR format and block partition ranges are recorded in the block file.

their out-edges into a block according to the ascending order of their IDs until the data volume in the block exceeds a predefined *block size*, and then we create a new block. Figure 6 shows the data layout of the example graph in Figure 1(a). Besides, this lightweight graph data organization decreases the storage cost of each subgraph, and thus reduces the time cost of graph loading. As GraphWalker partitions a graph by simply reading through the *index file* once to record the beginning vertex of each block, it is also flexible to adjust block size for different applications.

For setting the graph block size, we find that a trade-off exists. That is, using smaller blocks can avoid loading more data which are not needed for updating random walks, while using larger blocks can have more walks getting updated in each subgraph loading. Besides, different analysis tasks require different walk scales, and thus prefer different block sizes. Lightweight tasks with a small number of walks prefer a small block size as the I/O utilization can get improved under this setting. In contrast, heavyweight tasks with a large number of walks prefer a large block size as large block size can increase the walk updating rate. Based on this understanding, we use an empirical analysis (see §4.4), and set the default block size as $2^{(\log_{10} R+2)}$ MB, where R is the total number of random walks. For example, in the case of running one billion walks, the default block size is 2 GB, which is usually smaller than the memory capacity of a commodity machine, so it is easy to keep a graph block in memory.

Graph loading and block caching. GraphWalker converts the graph format and partitions graph blocks in preprocessing phase. During the phase of running random walks, GraphWalker chooses a graph block and loads it into memory according to the the states of walks, and in particular, it loads the block containing the largest number of walks. After finishing analysis over the loaded graph block, it then chooses another block to load in the same way.

To ease the impact of block size and improve cache efficiency, GraphWalker also enables block caching by developing a walk-conscious caching scheme to keep multiple blocks in memory. The rationale is that blocks with more walks are more likely to be needed again in near future. Thus, the graph loading process with block caching works as follows. As illustrated in Figure 7, we first select a candidate block based on the state-aware model, to load this block, we check whether it is cached in memory or not. If it is already in memory, then we directly access memory to perform analysis.

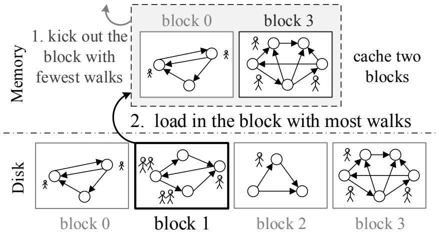

Memory
1. kick out the block with fewest walks
block 0
block 3
cache two blocks
2. load in the block with most walks
Disk
block 0
block 1
block 2
block 3

Figure 7: State-aware graph loading with block caching

Otherwise, we load it from disk, and also evict out the block in memory containing the fewest walks if the cache is full. The maximum number of blocks cached in memory depends on the usable memory size.

We emphasize that this walk-conscious block caching scheme differs from conventional page cache in the following aspects. First, we do not adopt prefetching as the state-aware model does not prefer to access graph blocks sequentially. Second, page cache manages data in memory at a *page* granularity, while we manage at a *block* granularity so as to fit the block-based graph loading and computing. Last but not least, the eviction policy also leverages walk states, which is different from LRU. Our experiments show that the walk-conscious block caching scheme always outperforms conventional page cache scheme.

3.3 Asynchronous Walk Updating

Note that in iteration-based systems, after loading a graph block, each walk in the loaded subgraph walks only one step, which induces to very low walk updating rate. In fact, after walking one step, many walks are still staying at the vertices in the current subgraph, so they can be further updated with more steps. To improve the I/O efficiency, some works use the *loaded data re-entry* [6], which allows the walks to reuse the loaded data. Lumos [42] uses cross-iteration value propagation to formalize reusing of loaded data for the subsequent iteration in order to provide synchronous guarantees. The idea is to re-enter the subgraph again to walk one more step by traversing the vertices in the subgraph again. Moreover, one can also keep re-entering the subgraph until all of the walks reach the boundary of the subgraph.

However, the re-entering scheme may cause local straggler problem. That is, many walks are able to move one step at the first time when the graph block was just loaded, and as the number of re-entries increases, most walks may reach the boundary of the subgraph, and only few walks remain in the subgraph, and they cost multiple re-entries to finish. Our experiments show that the last 20% of walks in a block may cost 60% of re-entries. These re-entries have very low utilization and cost a lot of time. We also find that simply stopping walking over the currently loaded subgraph after certain re-entries cannot address the local straggler problem either, and it does not reduce the completion time as the last few walks still remain in the subgraph and we still need to

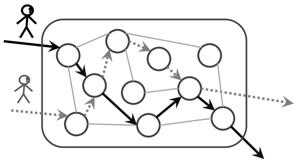


Figure 8: Asynchronous walk updating in parallel

re-load it with extra I/Os to finish the walks.

To further improve the I/O utilization and walk updating rate, GraphWalker adopts an asynchronous walk updating strategy, which allows each walk to keep updating until it reaches the boundary of the loaded graph block. After finishing a walk, we choose another walk to process until all walks in the current graph block are processed. Then we load another graph block based on the state-aware model described above. Figure 8 shows an example of processing two walks within the same graph block. To accelerate the computation, we also use multi-threading to update walks in parallel. We emphasize that with our asynchronous walk updating model, we completely avoid useless visits of vertices and eliminate the local straggler problem.

However, the state-aware model may lead to the *global straggler problem*. That is, some walks may move very fast and make a large progress as the graph data they needed can always be satisfied, while some other walks may move very slowly as they may be trapped in some coldblocks which are not loaded into memory for a long time. As a result, GraphWalker can quickly complete most walks, but takes a long time to finish the remaining few walks. Our experiments show that a few walks often incur nearly half of total I/Os.

To address the global straggler problem, we introduce a probabilistic approach into the state-aware graph loading process in GraphWalker. The idea is to give stragglers a chance to move some steps such that they can catch up the progress of most walks. Specifically, every time when we choose a graph block to load, we assign a probability p to choose the block containing walks with the slowest progress, i.e., with the smallest number of walked steps, and with probability $1-p$, we still load the block with the most walks. Note that the global straggler problem will be mitigated more efficiently as p increases, but the efficiency of the majority of walks will decrease. So there is a trade-off for setting p. Based on our empirical analysis, we find that $p = 0.2$ is an appropriate setting, and we can get 20% improvement in some cases.

3.4 Block-Centric Walk Management

We record each walk with three variables, source, current and step, which indicate the start vertex, the offset of the current vertex in the block, and the number of moved steps, respectively. We record each walk with 64 bits. The number of bits allocated for each variable is shown in Figure 9. This data structure can support starting random walks at 2^{24} source vertices simultaneously and it also allows each walk to move up to 2^{14} steps. Note that there is no limit on the total number of walks as we can start many walks at each vertex.

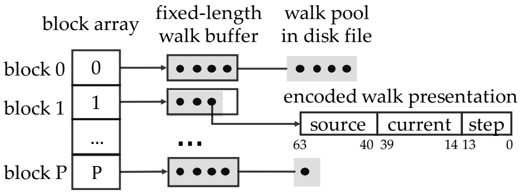


Figure 9: Block-centric walk management

To reduce the memory overhead of managing all walk states, we propose a block-centric scheme. For each graph block, we use a walk pool to record the walks which are currently in the block, referring to Figure 9. We implement each walk pool as a fixed-length buffer, which stores at most 1024 walks by default, so as to avoid dynamic memory allocation cost. When there are more than 1024 walks in a block, we flush them to disk and store them as a file called *walk pool file*. Note that we encode each walk with a 64-bit *long* data type, so each walk pool only costs 8 KB. This way, the memory cost for managing walk state is very low. For example, for running one billion walks in YahooWeb, GraphWalker costs only 800 KB if it uses 100 graph blocks. However, DrunkardMob costs more than 4 GB as each walk uses at least four bytes. Besides, these walks jump among the 11.2M dynamic arrays (refer to §2.2), and thus cause frequent memory re-allocation and bring extra time cost.

When we load a graph block into memory, we also load its walk pool file into memory and merge the walks with those stored in the in-memory walk pool. Then we perform random walks and update walks in current walk pool. During the update process, when a walk pool is full, we flush all walks in the walk pool to disk by appending them to the corresponding walk pool file and clear the buffer. When finish computing with the loaded graph block, we clear the current walk pool and sum up the walks in both walk buffer and walk pool file of each block so as to update the walk states.

With this lightweight walk management, we save a lot of memory cost for storing walk states, thus it is able to support massive concurrent walks. Besides, the fixed-length walk buffering strategy turns many small I/Os for updating walk states into several large I/Os, which largely reduces the I/O cost for providing persistent storage of walk states.

4 Evaluation

GraphWalker aims for providing fast and scalable random walks, so we take DrunkardMob [23], the state-of-the-art single-machine random walk specific graph system, as a baseline for performance comparison. Besides, there are also a number of single-machine graph systems, which further optimize the system performance from different aspects. For completeness, we also compare GraphWalker with two state-of-the-art graph systems Graphene [29] and GraFSoft [20]. To further validate its scalability, we compare GraphWalker with the most recent distributed random walk graph system, i.e., KnightKing [46].

4.1 Experiment Settings

Testbed. All experiments are performed on a Dell Power Edge R730 machine with 64GB memory and 24 Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors. The entire graph data are stored on a 3.2TB RAID-0 consisting of seven 500GB SamSung 860 SSDs if we do not state specifically. We also study the performance of GraphWalker on HDDs. For the distributed system KnightKing [46], it is run on an 8-node cluster with 10Gbps Ethernet inter connection, each node is equipped with two 8-core Intel Xeon E5-2620 v4 processors with 20MB L3 cache and 64GB DRAM.

Dataset. Table 1 lists the statistics of the six graph datasets we used. TT [4], FS [1], YW [5] and CW [3] are real-world graphs. K30 and K31 are two synthetic graphs generated with Graph500 kronecker [2]. These graphs are all widely used in graph system evaluations. CSR Size indicates the minimum storage cost by storing graphs in CSR format, and Text Size is the size of the dataset stored in text format as an edge list. We point out that Kron30, Kron31 and CW are large graphs that can not be entirely put into the memory in our testbed, and CW is the largest web corpus available in public.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	CSR Size	Text Size
Twitter (TT)	61.6M	1.5B	6.2GB	26.2GB
Friendster (FS)	68.3M	2.6B	10.7GB	47.3GB
YahooWeb (YW)	1.4B	6.6B	37.6GB	108.5GB
Kron30 (K30)	1B	32B	136GB	638GB
Kron31 (K31)	2B	64B	272GB	1.4TB
CrawlWeb (CW)	3.5B	128B	540GB	2.6TB

Table 1: Statistics of Datasets

Graph algorithms. Besides directly evaluating the performance of running random walks, we also consider the following four common random walk based algorithms.

- *Random Walk Domination (RWD)* [27]. We start one walk of length six from each vertex in the graph to find a vertex set which has the maximum influence diffusion.
- *Graphlet Concentration (Graphlet)* [34, 35]. We use a special graphlet, triangle, as a study case. We randomly start 100 thousand random walks of length four to estimate the ratio of triangles in the graph.
- *Personalized PageRank (PPR)* [12]. We simulate 2000 random walks of length 10 starting at each query source vertex to approximate the PPR, which was shown to be sufficient to ensure the accuracy.
- *SimRank (SR)* [19]. We start 2000 random walks of length 11 respectively from the query pair vertices to compute the expected meeting time,

Remark. The first two are graph computation algorithms which utilize the entire graph, while the other two are graph query algorithms which need only a portion of the graph. We point out that all of them are classical and representative graph algorithms. We run each experiment ten times and compute the average completion time. Before each execution, we also clear the page cache to avoid its impact.

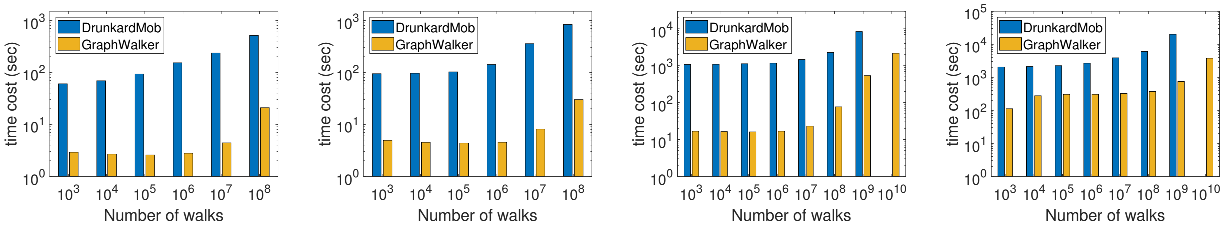
4.2 Comparison with RW-Specific Systems

We validate the efficiency of GraphWalker by comparing it with DrunkardMob, the state-of-the-art single-machine system that is specially optimized for random walk. Both GraphWalker and DrunkardMob are implemented based on GraphChi. Note that random walk based algorithms usually require to start certain number of random walks with certain walk length, so we evaluate the performance by considering different random walk configurations, so as to study the performance of the entire design space and demonstrate the scalability of GraphWalker in supporting large amount of random walks with very long walk length. We also show the performance of the four random walk based algorithms. Finally, we justify the improvement achieved by GraphWalker by using micro-benchmark results.

4.2.1 Performance Study in Entire Design Space

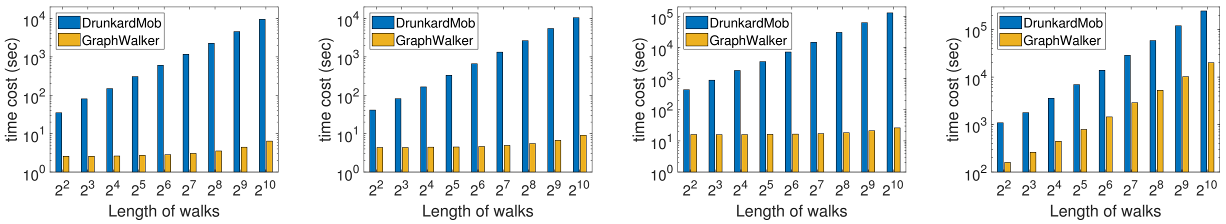
We first show the results by fixing the walk length to ten, but varying the number of walks from 10^3 to 10^{10}, as depicted in Figure 10. In each figure, the x-axis indicates the number of walks configured in each experiment, and the y-axis shows the time needed to finish running all these walks. First, we can see that GraphWalker is consistently faster than DrunkardMob under all settings for different numbers of walks and different graph datasets. In particular, in the case of running 10^6 walks on YahooWeb, DrunkardMob costs near 20 minutes, while GraphWalker takes only 17.8 seconds. That is, GraphWalker achieves 70× speedup. In general, GraphWalker achieves 16× to 70× speedup under all settings. In addition, for the case when the number of walks is not too large, then I/O cost is a dominate factor, so the total time of running different number of walks is almost a constant. However, as the number of walks continues to increase, computation cost becomes larger, so the total time cost also increases linearly when we run more random walks.

One attractive feature of GraphWalker we like to highlight is its scalability. We point out that even for running tens of billions of random walks on large graphs, GraphWalker can still finish within a reasonable time. However, DrunkardMob even fails to run 10^{10} walks on large graphs, due to the out-of-memory error, so we do not show the results of DrunkardMob in the setting of more than 10^{10} walks. More importantly, when the graph becomes really large, DrunkardMob may fail to run. For example, for Kron31 and CrawlWeb, DrunkardMob also encounters the out-of-memory error. Thus, we do not show the results on them for the interest of space. The main reasons are as follows: (1) DrunkardMob keeps all walk states in memory, so it's hard to support massive walks. (2) DrunkardMob employs a dynamic array index for every 128 vertices, so it incurs a large memory overhead when the graph becomes really large, and its also hard to write the walks to



(a) Twitter (b) Friendster (c) YahooWeb (d) Kron30

Figure 10: Performance of random walks with different number of walks by fixing walk length as 10.



(a) Twitter (b) Friendster (c) YahooWeb (d) Kron30

Figure 11: Performance of random walks with different walk lengths by fixing the number of walks as 10^5.

disk for too many open files needed. However, because of the block-centric walk indexing design and keeping walk states on disk, GraphWalker is capable to support huge graphs, e.g., Kron31 and CrawlWeb, even for running tens of billions of random walks, e.g., GraphWalker finishes running 10^{10} walks on CrawlWeb within around one hour.

We also evaluate the performance by varying the walk length. Here we fix the number of walks as 10^5 and vary the length of each walk from 2^2 to 2^{10}. The results are shown in Figure 11. First, we can see that GraphWalker is always much faster than DrunkardMob, and it achieves even more than three orders of magnitude in the best case. In particular, when the graph is not extremely large, e.g., for Twitter, Friendster, and YahooWeb, the time cost of DrunkardMob continues to increase when running longer walks, while that of GraphWalker is almost a constant, this is because GraphWalker can cache almost the whole graph in memory for medium-sized graphs, due to the lightweight block storage and optimized block catching strategy, and thus incurs very low I/O cost. For very large graphs which can not be fully put in memory, e.g., Kron30, the time cost of both DrunkardMob and GraphWalker increases as walks get longer, as GraphWalker needs to swap in and kick out blocks between memory and disk in this case. However, we point out that GraphWalker is much faster, e.g., it achieves 7× to 10× speedup even for Kron30. This experiment also demonstrates the scalability of GraphWalker in supporting long random walks which have thousands of steps.

4.2.2 Performance of Random Walk based Algorithms

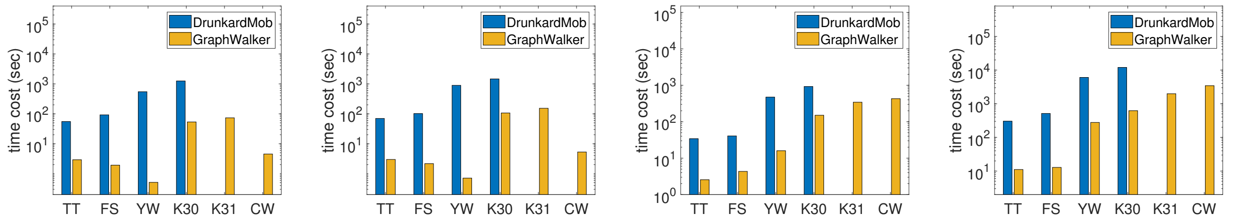
We now evaluate the performance of the four common random walk based algorithms described in §4.1. From Figure 12, we can see that GraphWalker achieves 9× to 48× speedup upon DrunkardMob. In particular, in some special cases, e.g., running PPR and SR on YahooWeb, GraphWalker even achieves more than three orders of magnitude speedup, this is because YahooWeb has a very good locality at the query vertices, so GraphWalker only needs to load several corresponding subgraphs to run random walks. However, DrunkardMob needs to iteratively scan the entire graph and updates walks in a synchronized manner, so it has a very low I/O utilization and takes long time.

We like to point out that DrunkardMob again fails to handle the two largest graphs. due to the same reason explained in §4.2.1, so we skip the results in these cases. Note that this experiment also demonstrates the scalability of GraphWalker in supporting massive walks and huge graphs.

4.2.3 Micro-benchmarks

Recall that the inefficiency of existing systems in running random walks mainly come from the iteration-based I/O model, and thus they suffer from low I/O utilization and low walk updating rate (refer to §2). To better understand why GraphWalker could significantly improve the overall performance as presented in the last subsection, we further consider these two micro-benchmarks to show how the state-aware I/O model in GraphWalker address the limitations. We also show the time cost breakdown to see how GraphWalker improves the performance of each part along the random walk process. *We only show the results of running RWD algorithm on YahooWeb, and results are similar for other settings.*

I/O utilization. I/O utilization is defined as the edge usage amount for updating walks divided by the total number of edges loaded by one I/O. Note that an edge may be reused by different walks, so we sum up the total times of being used for all edges. Thus, the I/O utilization defined here may exceed



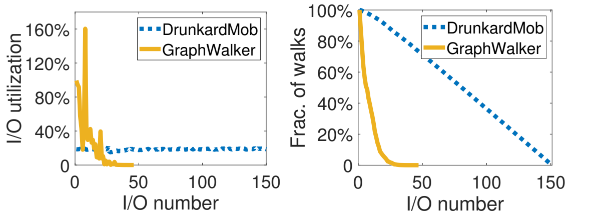
(a) Personalized PageRank (b) SimRank (c) Graphlet (d) Random Walk Domination

Figure 12: Performance of random walk based algorithms.

100% when one edge is used by multiple walks. We point out that DrunkardMob partitions the YahooWeb graph into 25 shards, and the walk length is six, so the total number of I/Os required by DrunkardMob is 150. We can see that the I/O utilization is only around 20% as shown in Figure 13(a). In contrast, GraphWalker needs only 46 I/Os to complete all walks, so the number of I/Os is significantly reduced. The I/O utilization of GraphWalker is also much higher than that of DrunkardMob. Specifically, the utilization of the first few I/Os reaches up to 80%-160%, this is because the subgraphs loaded by the first few I/Os have the most walks, and many of them may use more than one edge to update. Even for most I/Os, the I/O utilization of GraphWalker is between 40% to 80%, which is 2× to 4× compared to that of DrunkardMob.

Walk updating rate. Now we study the walk updating rate, shown in Figure 13(b). Recall that DrunkardMob updates 50 million steps per I/O on average and costs 150 I/Os to finish the computation. While GraphWalker significantly improves the walk updating rate, it only needs 46 I/Os to complete all walks and updates 185 million steps per I/O on average, which is 3.7× higher than that of DrunkardMob. The main reason is that DrunkardMob adopts the iteration-based I/O model, it walks only one step for each walk when loading one block. In contrast, GraphWalker develops an asynchronous walk updating method to fully utilize the loaded graph data in memory (see §3.3), so each walk may move multiple steps over the subgraph loaded by each I/O. As a result, GraphWalker saves a lot of I/Os and completes all random walks more quickly than DrunkardMob.

Time cost breakdown. To better understand the effect of the design optimizations in GraphWalker, we also show the time cost breakdown in Table 2. Note that in the whole execution procedure, there are three key operations: (1) graph loading, which loads graph blocks into memory with disk I/Os, (2) walk updating, which updates the walk states maintained in memory, and (3) walk persisting, which includes to read walk states from disk into memory and write back updated states to disk for persistency. Besides, the three operations are proceeded in an interleaved way, so we aggregate the total time of executing each operation. From the results, we can see that GraphWalker outperforms DrunkardMob in all aspects. The improvement is achieved by the integration of multiple design optimizations, which all contribute to the high efficiency of GraphWalker, e.g., the improvement of graph loading performance mainly comes from the state-aware scheme with the lightweight data organization and block caching policy, and it also benefits from the asynchronous updating strategy.



(a) I/O utilization (b) Walk updating rate

Figure 13: I/O utilization and walk updating rate (DrunkardMob needs 150 I/Os and GraphWalker only needs 46 I/Os)

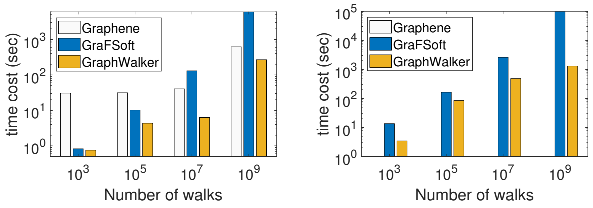
4.3 Comparison with State-of-the-art Systems

Single-machine graph systems. There are a number of optimizations being proposed in recent single-machine graph systems, e.g., fine-grained block partition, asynchronous I/O to support pipeline between I/O and computation, huge page support to reduce TLB miss, etc. These optimizations are not specific for random walks, so many of them are also orthogonal to the optimizations in GraphWalker. Thus, to further demonstrate the efficiency of GraphWalker, we also compare it with two state-of-the-art open-source single-machine systems, Graphene [29] and GraFBoost [20]. For fair comparison, we only focus on the pure software implementation of GraFBoost called GraFSoft. Note that GraphWalker is implemented based on the baseline system GraphChi, so it does not include the above mentioned design optimizations.

In this experiment, we focus on the case of running random walks starting from a single source due to page limit. We fix the walk length as ten and vary the number of walks. Note that Graphene is a semi-external system which stores graph data on disk while keeps all walk states in memory, so it is unable to handle the case of massive walks, e.g., greater than 10^9 walks, or large graphs, e.g., larger than Friendster, due to its high memory cost. In the interest of space, we only show

Time cost (s)	DrunkardMob	GraphWalker	Speedup
Graph Loading	1005	47	21×
Walk Updating	3029	214	14×
Walk Persisting	1056	16	66×
Total Runtime	5110	278	18×

Table 2: Time cost breakdown



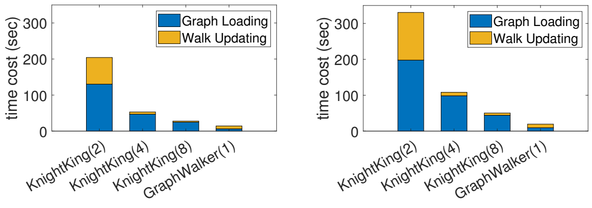
(a) Friendster (b) CrawlWeb

Figure 14: Comparison with Graphene and GraFSoft

the results for Friendster, the largest graph that Graphene can process, as well as the largest graph CrawlWeb. We observe similar results for other graphs.

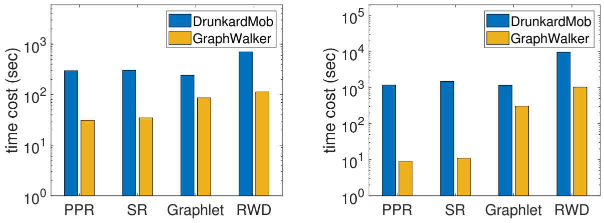
The results are shown in Figure 14, and we can have several conclusions. First, GraphWalker consistently outperforms Graphene even though Graphene is a semi-external system which does not require I/Os to write back walk states, and it achieves up to 19× speedup. More importantly, GraphWalker is also scalable to run huge amount of walks, as well as process extremely large graphs, while Graphene fails to run in these cases due to its high memory cost caused by the semi-external design. Second, compared with GraFSoft, when the number of walks is small, the improvement of GraphWalker is limited, because each block can only have a few walks given the small total number and the state-aware I/O model can not bring too much benefit. However, the improvement of GraphWalker increases as the number of random walks gets larger.For example, when running one billion random walks on CrawlWeb, GraphWalker spends only 21.8 minutes while GraFSoft can not even complete the task within 24 hours. That is, GraphWalker achieves at least 40× speedup. More importantly, as we increase the number of walks, the increase of time for GraphWalker is *sub-linear* and much slower than that of GraFSoft, this further demonstrates the scalability of GraphWalker in supporting huge amount of random walks.

Distributed random walk system. To further demonstrate the scalability of GraphWalker, and its resource-friendly feature, we also compare it with a distributed graph system, KnightKing [46], which is the most recent distributed graph system optimized for random walks. In the interest of space, we focus on a random walk based algorithm which is also used in KnightKing, i.e., PPR. Specifically, it starts one walk at each vertex, and each walk terminates with probability t in each step. We set $t = 0.15$, which is a very common setting in various applications [12, 25]. Note that smaller t means larger average number of walk steps and requires more computations, so KnightKing uses a small t to demonstrate its computing efficiency. As KnightKing uses a cluster of eight machines for evaluation in its paper, to enable cross-validation, we also use eight machines at most, and focus on the largest two graphs that can be handled by KnightKing with eight machines, i.e., Twitter and Friendster. We also convert the two graphs to be undirected as in KnightKing.



(a) Twitter (b) Friendster

Figure 15: Comparison with KnightKing



(a) Friendster (b) YahooWeb

Figure 16: Performance on HDDs

Figure 15 shows the results, and the number after each system name denotes the number of machines being used. We can see that for KnightKing, as the cluster size increases, the computing time, i.e., the time for updating walks, gets reduced greatly, but it still costs a lot of time for processing I/Os, i.e., loading graph blocks. This is because KnightKing mainly focuses on optimizing the computing efficiency, but not disk I/Os. Note that the results of the computing time in this experiment are consistent with those in the KnightKing paper. In contrast, GraphWalker mainly targets for the I/O efficiency problem, and also adapts the walk updating process accordingly based on its I/O model, so it can realize very fast random walks over disk-resident graphs. Here the walk updating time also incudes the time for walk index persistency. We also see that GraphWalker achieves comparable performance even compared with KnightKing running on eight machines. Even more, for the largest graph CrawlWeb, KnightKing may need a larger cluster to run according to the estimation of its used resources when processing other smaller graphs. Thus, we can conclude that GraphWalker is also a more resource-friendly alternative.

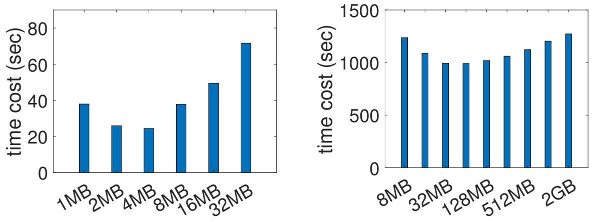
4.4 Impact of System Configurations

Performance on HDDs. We also study the impact of storage devices by running experiments on hard disk drives (HDDs). Figure 16 shows the time cost of running the four algorithms we considered in this paper, and we only show the results for Friendster and YahooWeb here. Since HDDs have much lower random I/O performance than SSDs, the time cost of both DrunkardMob and GraphWalker is increased. When comparing GraphWalker with DrunkardMob, we observe similar results as in the case of SSDs studied before. Precisely, GraphWalker achieves 3× to 135× speedup under different settings.

Impact of block size. In GraphWalker, block size has an impact on both I/O utilization and walk updating rate.

Specifically, smaller block size improves the I/O utilization, while larger blocks can make walks move more steps for each single I/O and thus improves the walk updating rate. To study the impact of block size, we keep only one block in memory and run the PPR algorithm on CrawlWeb as a study case. Here, we consider two PPR algorithms which start random walks from 1000 and 100000 sources, respectively. Note that the two cases are representative to denote two typical scenarios of accessing only a part of the graph or accessing most of the entire graph.

Figure 17 shows the results. We find that it necessitates an appropriate block size setting to achieve the best performance due to the above analyzed tradeoff. The insight is that small blocks may be beneficial to lightweight tasks which require only a small number of random walks, as the I/O utilization can get improved under this setting. In contrast, large blocks may be beneficial to heavyweight tasks which require a large number of random walks, as large block setting increases the walk updating rate. Based on this observation, we also propose a method to set the block size, which is determined according to the number of walks (see §3.2).



(a) PPR with 10^3 sources (b) PPR with 10^5 sources

Figure 17: Impact of block size

5 Related Work

A number of graph systems have been proposed in recent years, and some of them develop distributed systems based on a cluster of machines so as to handle very large graphs which can not reside on a single machine [8, 9, 13, 14, 21, 30, 32, 40, 48]. However, distributed graph systems usually require efficient graph partition and low-cost communication between machines. Besides, research efforts are also made to leverage large memory in analyzing large graphs [16, 33, 39] or utilizing GPUs to accelerate the computation [22, 28, 45].

Graph processing on single machine for disk-resident graphs also receives a lot of attentions. GraphChi [24] is the pioneering work, and X-Stream [37] further develops a different computation model based on edge streams. GridGraph [49] optimizes I/Os by selectively loading needed graph blocks to bypass useless graph data. DynamicShards [43] and Graphene [29] also aim to reduce the loading of useless edges by dynamically adjusting the graph partition layout. CLIP [6] and Lumos [42] improve the utilization of the loaded graph blocks so as to reduce the number of I/Os. There are also a body of works to leverage high-performance emerging devices to improve performances [10, 11, 20, 31]. The above systems are not designed specially for random walks, so most of them still follow the iteration-based model. Different from them, GraphWalker targets for supporting massive concurrent random walks in a fast and scalable way, and its key idea is to utilize the states of walks to optimize the process of graph loading and computing so as to improve the I/O and computing efficiency.

In terms of random walk, besides developing new algorithms, such as random walk with restart [41], FolkRank [17] and TrustWalker [18], etc., there are also some works focusing on system design. To support massive random walks on large graphs, DrunkardMob [23] proposes an encoded representation and a lightweight efficient index so as to be able to run billions of random walks on a single machine. As it follows the iteration-based model, it still suffers from the I/O deficiency problem. Different from DrunkardMob, GraphWalker focuses on optimizing the I/O management, and it develops a new state-aware model with asynchronous walk updating to improve I/O performance. In addition, GraphWalker also allows walk states to be stored on disk, instead of putting only in memory as in DrunkardMob, so it is more scalable to run more walks on larger graphs.

Besides single-machine random walk systems, there is also a distributed system KnightKing [46], which is recently proposed and also optimized for random walks. It provides a unified framework to support various random walks and focuses on optimizing the walking process without addressing disk I/Os. Different from KnightKing, GraphWalker mainly targets for addressing the I/O problem, and it is also more resource friendly as it can process massive random walks on large graphs on just a single machine.

6 Conclusion

In this paper, we proposed GraphWalker which is an I/O-efficient system for supporting fast and scalable random walks over large graphs on a single machine. GraphWalker carefully manages graph data and walk indexes, and optimizes I/O efficiency by using state-aware graph loading and asynchronous walk updating. Experiment results on our prototype show that GraphWalker outperforms state-of-the-art single-machine systems, and it also achieves comparable performance with distributed graph system running on a cluster machine. In the future work, we will consider to extend the state-aware design idea in GraphWalker to distributed clusters so as to process massive analytic tasks in parallel.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their comments. This work is supported in part by the National Key R&D Program of China (2018YFB1800203), NSFC (61772484), Youth Innovation Promotion Association CAS, and USTC Research Funds of the Double First-Class Initiative (YD2150002003). Yongkun Li is USTC Tang Scholar, and he is the corresponding author.

References

[1] Friendster. http://konect.uni-koblenz.de/networks/friendster.

[2] Graph500. https://graph500.org/.

[3] The 2012 common crawl graph. http://webdatacommons.org.

[4] Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[5] Yahoo Webscope Program. http://webscope.sandbox.yahoo.com.

[6] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing Out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC*, 2017.

[7] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. Approximating Aggregate Queries about Web Pages via Random Walks. In *VLDB*, 2000.

[8] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: an Efficient Task-Oriented Graph Mining System. In *ACN EuroSys*, 2018.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys*. ACM, 2015.

[10] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-node Graphs on an Array of Commodity SSDs. In *USENIX FAST*, 2015.

[11] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-Scale Graph Processing on Emerging Storage Devices. In *FAST*. USENIX, 2019.

[12] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards Scaling Fully Personalized Pagerank: Algorithms, Lower Bounds, and Experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. USENIX, 2012.

[14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. USENIX, 2014.

[15] Monika R Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc Najork. Measuring Index Quality using Random Walks on the Web. *Computer Networks*, 31(11):1291–1303, 1999.

[16] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for Easy and Efficient Graph Analysis. In *Proceedings of the ACM SIGARCH Computer Architecture News*, 2012.

[17] Andreas Hotho, Robert Jäschke, Christoph Schmitz, Gerd Stumme, and Klaus-Dieter Althoff. Folkrank: A Ranking Algorithm for Folksonomies. In *Lwa*, 2006.

[18] Mohsen Jamali and Martin Ester. Trustwalker: a Random Walk Model for Combining Trust-Based and Ttem-Based Recommendation. In *ACM SIGKDD*, 2009.

[19] Glen Jeh and Jennifer Widom. SimRank: a Measure of Structural-context Similarity. In *ACM SIGKDD*, 2002.

[20] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *ISCA*. IEEE, 2018.

[21] Arijit Khan, Gustavo Segovia, and Donald Kossmann. On Smart Query Routing: for Distributed Graph Querying with Decoupled Storage. In *ATC*. USENIX, 2018.

[22] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.

[23] Aapo Kyrola. Drunkardmob: Billions of Random Walks on Just a PC. In *ACM RecSys*, 2013.

[24] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale Graph Computation on Just a PC. In *OSDI*, 2012.

[25] Meyer C D. Langville A N. Deeper inside Pagerank. In *Internet Mathematics*, 2004.

[26] Chul-Ho Lee, Xin Xu, and Do Young Eun. Beyond Random Walk and Metropolis-hastings Samplers: Why You Should Not Backtrack for Unbiased Graph Sampling. In *SIGMETRICS*, 2012.

[27] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. Random-walk Domination in Large Graphs. In *ICDE*. IEEE, 2014.

[28] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015.

[29] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*. USENIX, 2017.

[30] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *VLDB*, 2012.

[31] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-edge Graph on a Single Machine. In *EuroSys*. ACM, 2017.

[32] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*. ACM, 2010.

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*. ACM, 2013.

[34] Nataša Pržulj. Biological Network Comparison Using Graphlet Degree Distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[35] Nataša Pržulj, Derek G Corneil, and Igor Jurisica. Modeling Interactome: Scale-Free or Geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[36] Bruno Ribeiro and Don Towsley. Estimating and Sampling Graphs with Multidimensional Random Walks. In *SIGCOMM*, 2010.

[37] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP*. ACM, 2013.

[38] Paat Rusmevichientong, David M Pennock, Steve Lawrence, and C Lee Giles. Methods for Sampling Pages Uniformly from the World Wide Web. In *Proceedings of the AAAI Fall Symposium on Using Uncertainty Within Computation*, 2001.

[39] Julian Shun and Guy E Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN*, 2013.

[40] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *SOSP*. ACM, 2015.

[41] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast Random Walk with Restart and Its Applications. In *ICDM*. IEEE, 2006.

[42] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *ATC*. USENIX, 2019.

[43] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, 2016.

[44] Rui Wang, Min Lv, Zhiyong Wu, Yongkun Li, and Yinlong Xu. Fast Graph Centrality Computation via Sampling: a Case Study of Influence Maximisation over OSNs. *International Journal of High Performance Computing and Networking*, 14(1):92–101, 2019.

[45] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN*, 2016.

[46] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. KnightKing: A Fast Distributed Graph Random Walk Engine. In *SOSP*. ACM, 2019.

[47] Pengpeng Zhao, Yongkun Li, Hong Xie, Zhiyong Wu, Yinlong Xu, and John CS Lui. Measuring and Maximizing Influence via Random Walk in Social Activity Networks. In *International Conference on Database Systems for Advanced Applications*, pages 323–338. Springer, 2017.

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*. USENIX, 2016.

[49] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Grid-Graph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*, 2015.

Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling

Long Zheng[1] Xianliang Li[1] Yaohui Zheng[1] Yu Huang[1] Xiaofei Liao[1] Hai Jin[1]
Jingling Xue[2] Zhiyuan Shao[1] Qiang-Sheng Hua[1]
[1]*National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology*
[2]*UNSW Sydney*
{longzh, xianliang, yaohui, yuh, xfliao, hjin, zyshao, qshua}@hust.edu.cn; j.xue@unsw.edu.au

Abstract

We introduce Scaph, a GPU-accelerated graph system that achieves scale-up graph processing on large-scale graphs that are initially partitioned into subgraphs at the host to enable iterative graph computations on the subgraphs on the GPU. For active subgraphs to be processed on GPU at an iteration, the prior work always streams each in its entirety to GPU, even though only the neighboring information for its active vertices will ever be used. In contrast, Scaph boosts performance significantly by reducing the amount of such redundant data transferred, thereby improving the effective utilization of the host-GPU bandwidth drastically. The key novelty of Scaph is to classify adaptively at each iteration whether a subgraph is a high-value subgraph (if it is likely to be traversed extensively in the current and future iterations) or a low-value subgraph (otherwise). Scaph then schedules a sub-graph for graph processing on GPU using two graph processing engines, one for high-value subgraphs, which will be streamed to GPU entirely and iterated over repeatedly, one for low-value subgraphs, for which only the neighboring information needed for its active vertices is transferred. Evaluation on real-world and synthesized large-scale graphs shows that Scaph outperforms the state-of-the-art, Totem (4.12×), Graphie (8.93×), and Garaph (3.71×), on average.

1 Introduction

Graph processing is used in a variety of real-world applications, including path navigation [23], social network analysis [9], and financial fraud detection [27]. Graph processing, typically memory-bound, often benefits substantially from memory optimizations [50]. Compared to CPU-based graph systems [15, 16, 30, 36, 40, 46, 51, 68], GPU-accelerated graph systems can have high internal bandwidth and massive parallelism, therefore offering superior speedup [19, 25, 39, 66], even for graph algorithms that involve substantial light-weight integer and comparison-based operations [28].

Unfortunately, many real-world graphs still cannot fit into GPU memory to enjoy high-performance in-memory graph processing. For example, NVIDIA's high-end Tesla V100 has 32GB global memory [42], while real-world graphs such as `Facebook`'s can easily reach the terabyte-scale [9]. This gap has spurred the development of many distributed graph systems, which partition a graph into sub-graphs and then assign these sub-graphs to different machines for distributed computing [13, 15, 16, 33, 36, 67]. However, these distributed graph systems suffer from prohibitive communication overheads [8, 15, 58] and also require an extensive range of domain knowledge to maintain [11, 16, 24, 38, 56, 59].

	GTX980	K40	P100
Arch.	Maxwell	Kepler	Pascal
#SMXs	16	15	56
#Cores	2048	2880	3584
Memory	4GB	12GB	16GB
BW	224 GB/s	288 GB/s	720 GB/s

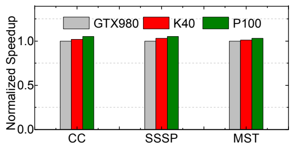


Figure 1: Performance of Graphie [17] for three representative graph algorithms on `fb-2009` (a graph with 139.1M vertices and 12.3B edges, taking 137.8GB (unweighted) and 275.6GB (weighted)) on three different generations of GPUs plugged (separately) in a 28-core host machine with 512GB memory

There is nowadays a viable alternative of turning a single machine plugged in with a GPU to support scale-up large-scale graph processing. Such a GPU-accelerated heterogeneous platform is easy to use and maintain [30, 62, 68]. In addition, we can take advantage of the large host memory (at the terabyte scale) to store large-scale graphs while still enjoying high-performance graph processing on GPU.

In this paper, we focus on building graph systems on GPU-accelerated heterogeneous platforms to achieve scale-up graph processing for large graphs that cannot fit into GPU memory. This would enable high-performance graph analytics on large-scale graphs everywhere by simply plugging a GPU into an off-the-shelf commodity PC. In this case, a large graph must be partitioned into subgraphs at the host. Any subgraphs to be processed on GPU must be streamed asynchronously to GPU when some previously transferred subgraphs are being concurrently processed on GPU (in an overlapping manner). We consider vertex-centric graph processing [36], where a graph algorithm is performed in a sequence of iterations until convergence [15,36]. In each iteration, a graph algorithm processes only the *active vertices* (vertices with ongoing updates) in each subgraph, updates their neighbors (along their outgoing edges) and activates the neighbors whose values have been updated. In this paper, we restrict ourselves to handle large-scale graphs that can entirely fit into the host memory. Meanwhile, all the vertex data, including vertex states (active or not), are assumed to be resident in the GPU memory. In contrast, the edge data of a graph are stored at the host and partitioned into subgraphs. During graph processing, *active subgraphs* (containing all out-going edges of an active vertex) must be transferred to GPU for iterative processing.

Achieving scale-up graph processing for large-scale graphs on GPU-accelerated heterogeneous platforms is challenging. The power-law graphs [15] can result in substantial load imbalance among threads and warps [39]. Irregular data accesses made in graph algorithms often lead to non-coalesced memory accesses for GPU graph processing. Fortunately, effective techniques for addressing these performance-limiting issues exist [14, 17, 34]. Currently, the performance bottleneck in a GPU-accelerated graph system has shifted to the limited host-GPU bandwidth, which was relatively sufficient in the past (e.g., ∼11.4GB/s for PCI-Express 3.0). However, existing graph processing engines [17, 26, 34, 47] focus still on overcoming the GPU memory capacity limitation to enable large-scale graph processing, without paying adequate attention to the effective utilization of the host-GPU bandwidth.

Simple heuristics are used to reduce the number of data transfers. Totem [14] partitions a graph into two subgraphs, one for the host and one for GPU, by keeping the amount of data transfers to a minimum at the expense of severe load imbalance. Garaph [34] concurrently processes all active subgraphs on both the host and GPU. Graphie [17] processes all subgraphs on GPU but re-processes only the recently processed subgraphs in the next iteration (before they are removed from GPU memory). However, these graph systems always transfer an active subgraph in its entirety to GPU (even though only the neighboring information for its active vertices will usually be used), resulting in poor utilization of the host-GPU bandwidth. To see this, Figure 1 compares the performance results of Graphie [17] for running three graph algorithms on a large graph on a PC with three generations of GPUs (one at a time). We see little performance gains when increasingly more powerful GPUs are used. For example, P100 has over 3× as many #SMX's and 4× as much memory as GTX980, but it offers small performance improvements.

Recently, hardware vendors have launched several advanced interconnect technologies to mitigate the impact of the "bandwidth wall". For example, compared to PCI-E 3.0, NVLINK 2.0 (50GB/s per link) and PCI-E 4.0 (32GB/s) are several times faster, but still cannot keep up with the growth in GPU computing capabilities. Specifically, these advanced technologies cannot yet provide ∼500GB/s required by graph analytics under existing computing platforms [1].

In this work, we argue that we can improve the performance of large-scale graph processing on a GPU-accelerated architecture significantly by improving the effective utilization of the host-GPU bandwidth. Our key observation is that the majority of the data in an active subgraph (once streamed to GPU) are never used in current and future iterations (§2.2). We introduce Scaph that achieves significantly improved performance than state of the art by adopting value-driven differential scheduling for active subgraphs. The key novelty is to classify an active subgraph adaptively into a *high-value subgraph* (if it will be extensively traversed in current and future iterations) and a *low-value subgraph* (otherwise). Thus, a high-value subgraph contains a significant amount of *useful data (UD)* to be used by active vertices in the current iteration and of *potentially useful data (PUD)* to be used by its future active vertices in future iterations. On the other hand, a low-value subgraph contains a lot of *never-used data (NUD)* in current and future iterations.

Unlike earlier graph systems [17,26,34,47], which transfer an active subgraph to GPU in its entirety (but with only its UD used usually), Scaph uses the host to stream an active sub-graph to GPU by using two graph processing engines for handling high-value and low-value subgraphs, respectively. For the high-value subgraph, it will be transferred to GPU entirely. Inspired by the data movement reduction in out-of-core settings [2, 53, 69], we propose to compute each high-value subgraph multiple times to exploit its PUD ahead of schedule for accelerating convergence. Unlike these earlier efforts focusing on exploiting only the PUD in a subgraph, we present a GPU-context-friendly delayed scheduling to enable exploiting the PUD *across subgraphs on GPU* such that the value of the high-value subgraphs can be maximized. For the low-value subgraph, only the neighboring information for its active vertices is transferred and scheduled once.

In summary, this paper makes the following contributions:

- *Subgraph Value Characterization.* We quantify the value of a subgraph adaptively (dynamically) in terms of its UD and PUD used in current and future iterations.
- *Value-Driven Differential Scheduling.* We propose a scheduler that adaptively distinguishes high- and low-value subgraphs in each iteration and dispatches a subgraph to an appropriate graph processing engine for acceleration.
- *Value-Driven Graph Processing Engines.* We introduce two graph processing engines to squeeze the most value out of high- and low-value subgraphs to maximize the effective utilization of the host-GPU bandwidth in each case.
- *Evaluation.* We evaluate Scaph on both real-world and synthesized large graphs. Scaph outperforms state-of-the-art heterogeneous graph systems, Totem (4.12×) [14], Graphie (8.93×) [17], and Garaph (3.71×) [34], on average.

The rest of this paper is organized as follows. §2 describes the background and motivation. §3 gives an overview of

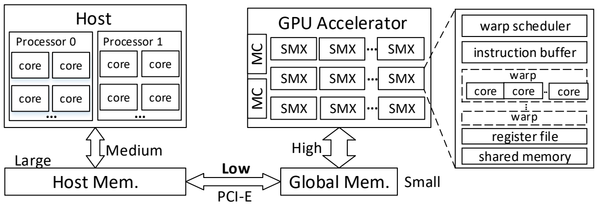


Figure 2: A GPU-accelerated heterogeneous architecture

Scaph. §4 describes value-driven differential scheduling while §5 discusses how to accomplish this effectively. §6 presents results. §7 discusses the related work. Finally, §8 concludes.

2 Background and Motivation

We first review the background. We then present some case studies to reveal why the poor host-GPU bandwidth utilization has limited the performance achieved by existing heterogeneous graph systems, finally motivating Scaph.

2.1 Host-GPU Heterogeneous Architectures

Figure 2 shows a representative GPU-accelerated heterogeneous architecture that integrates the hardware advantages of the host (with a larger host memory) and the GPU (with a stronger computing ability). A GPU consists of multiple *streaming multiprocessors* (SMXs), each of which includes hundreds of cores. Compared to the high-speed internal bandwidth (e.g., ∼720GB/s for NVIDIA Tesla P100 [41]) of GPU cores accessing global memory, a GPU is generally connected to the host with a relatively slow interface. For example, the host-GPU bandwidth via PCI Express 3.0 can be limited to be as low as ∼11.4GB/s in practice [5]. This significant performance gap often severely limits the performance potential achieved on a GPU-accelerated heterogeneous architecture if the host-GPU data transfers are frequent [17, 26]. This work makes use of a PCI Express interconnect since it is commonly used in the current commodity market.

2.2 A Motivating Study

Existing heterogeneous graph systems [17, 26, 47], with Graphie [17] as a representative compared against in our evaluation, generally use host memory to store large-scale graphs (partitioned into subgraphs) and rely on GPUs exclusively to accelerate graph analytics on these subgraphs. Figure 3 depicts their generic graph processing engine used, with the function calls in blue executed on GPU. Due to the limited GPU memory, a graph G is first divided into subgraphs, $\tilde{G}_1, \cdots, \tilde{G}_n$ (line 2). During the entire iterative graph processing, the vertex data of G always reside in GPU memory, but the edge data of G, which are spread across these subgraphs, will be streamed to GPU on-demand [17, 26, 34, 47].

At each iteration (lines 5 – 12), $\tilde{G}_{active}$ represents the set of active subgraphs, i.e., the ones containing some out-going edges of an active vertex. In each iteration, all active vertices are processed. If their out-going edges are not in the GPU, their containing (active) subgraphs are transferred to GPU in their entirety. Afterward, these active vertices will be processed on the GPU (lines 13 – 18) to activate more destination vertices possibly. Note that Graphie [17] may schedule first the subgraphs processed at the end of the previous iteration as they are still in GPU memory (line 8).

```
1   Procedure SimpleSubgraphEngine(Graph $G$)
2       Load $\tilde{G}$'s subgraphs in $\{\tilde{G}_1, \cdots, \tilde{G}_n\}$ into the host
3       VertexInitialization($G$)
4       $\tilde{G}_{active} \leftarrow$ FindActiveSubgraph($G$)
5       while $\tilde{G}_{active} \neq \emptyset$ do
6           foreach $\tilde{G}_i \in \tilde{G}_{active}$ do
7               $stream \leftarrow$ DispatchStream($\tilde{G}_i$)
8               if $\tilde{G}_i$ is not resident in GPU memory then
9                   $GBuf \leftarrow$ AllocateDeviceMemory( )
10                  TransferData($stream$, $GBuf$, $\tilde{G}_i$, CPU2GPU)
11              Kernel ($stream$, $\tilde{G}_i$)
12          $\tilde{G}_{active} \leftarrow$ FindActiveSubgraph($G$)
    /* Graph Processing Kernel on the GPU */
13  Procedure Kernel(Subgraph $\tilde{G}$)
14      foreach $v \in \tilde{G}.SetOfVertices$ do
15          if $v$ is active then
16              foreach $e \in v.outedges$ do
17                  if Update($v$, $e$) = SUCCESS then
18                      Activate($e.destination\_vertex$)
```

Figure 3: Existing graph processing engine on a GPU-accelerated heterogeneous architecture (with the function calls in blue executed on GPU and all the rest on the host)

Table 1: The amount of used/unused data in the subgraphs transferred to GPU

	Algo.	Used	Unused
TW	CC	12.15GB	21.44GB
	SSSP	22.74GB	77.42GB
	MST	25.78GB	106.47GB
UK	CC	43.41GB	688.43GB
	SSSP	81.64GB	1302.85GB
	MST	134.97GB	2099.25GB

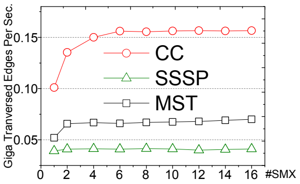


Figure 4: Performance of Graphie for TW with different number of SMXs

This simple graph processing engine does not effectively utilize the limited, scarce host-GPU bandwidth since many vertices in an active subgraph are not active. Simply transferring an entire subgraph to GPU (line 10) but consuming only a fraction of its data (lines 14 – 15) will waste a considerable amount of the host-GPU bandwidth. As a result, all the required data cannot arrive at the GPU promptly, limiting the performance that can be potentially achieved on GPU.

Let us examine the ratios of the unused over used data in the subgraphs transferred to GPU for three graph algorithms operating on two graphs, twitter (TW) [29] and uk-2007 (UK) [6], by Graphie [17] using the graph processing engine given in Figure 3. Table 1 gives the results obtained through an offline trace analysis, showing that these ratios range from 6.29 to 36.17. This indicates that the host-GPU bandwidth under Graphie is utilized rather ineffectively. Consequently, as shown further in Figure 4, the performance of Graphie for

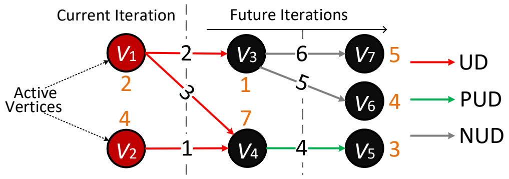
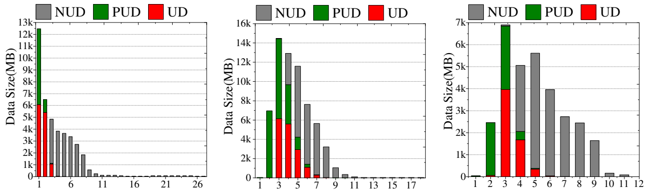


Figure 5: UD, PUD, and NUD in a subgraph, which may change across the iterations, illustrated for SSSP. The weight of an edge denotes its distance. The shortest distance found so far by SSSP at a vertex is depicted next to it in orange.



(a) CC

(b) SSSP

(c) MST

Figure 6: The amount of UD, PUD, and NUD across the iterations for three graph algorithms on twitter (TW) [29]

each graph algorithm (operating on TW) has plateaued as soon as #SMXs = 4. However, mainstream GPU accelerators usually have far more than 4 SMXs. For example, NVIDIA's Tesla K80 has 26 SMXs, while P100 has been integrated with 56 SMXs. Thus, a significant gap remains between the poor provision of data and high-speed computation of GPU.

2.3 Value-Driven Subgraph Scheduling

For a subgraph, its active vertices vary across the iterations. However, from the perspective of an active vertex, it always contains three types of edge data, as illustrated in Figure 5:

- *Useful Data (UD).* These are the edge data associated with all the active vertices in a subgraph, i.e., $V_1 \xrightarrow{2} V_3$, $V_1 \xrightarrow{3} V_4$, and $V_2 \xrightarrow{1} V_4$ in Figure 5. UD will definitely be used in the current iteration (lines 15 – 16 in Figure 3) and must be transferred to GPU [17, 26, 47].
- *Potentially Useful Data (PUD).* These are the edge data associated with all the future active vertices in future iterations in a subgraph. In Figure 5, PUD will be just $V_4 \xrightarrow{4} V_5$, since V_4 will be the only one activated by both V_1 and V_2 in current and future iterations. Unlike UD, PUD is not actually used in the current iteration, but may be transferred repeatedly to GPU if not handled carefully (as in the case of Figure 3 where PUD is usually discarded).
- *Never Used Data (NUD).* These are the edge data that will never be used again in a subgraph, associated with its vertices that have converged and will thus never be active. In Figure 5, NUD are $V_3 \xrightarrow{5} V_6$ and $V_3 \xrightarrow{6} V_7$.

Note that the same vertex may be activated many times in different iterations. Given a subgraph, its UD, PUD, and NUD computed at different iterations can vary dynamically.

Figure 6 shows the amount of UD, PUD, and NUD for the

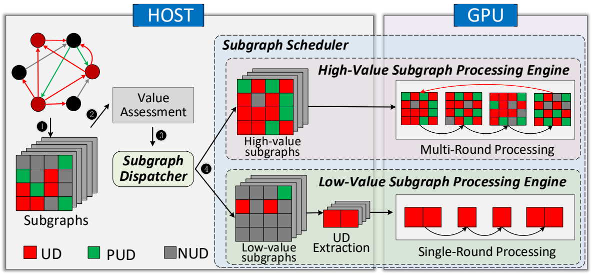


Figure 7: The workflow of Scaph

active subgraphs across all the iterations for three graph algorithms operating on twitter (TW) [29], partitioned sequentially into subgraphs of 32MB each. Graphie [17], a representative of existing heterogeneous graph systems [17, 26, 47], wastes the host-GPU bandwidth in two ways (Figure 3). First, PUD, usually discarded by Graphie but needed in future iterations, is substantial in earlier iterations. Second, NUD, which is becoming increasingly more dominant as the iteration progresses, is streamed to GPU redundantly.

For a subgraph, it will be cost-ineffective to stream just its UD, since its PUD cannot be exploited simultaneously. Instead, our key insight for improving the effective utilization of the host-GPU bandwidth is to look beyond the current iteration, by considering not only its UD in the current iteration but also its PUD in future iterations. Based on a cost-benefit analysis, we aim to leverage rather than discard its PUD (once streamed to GPU) in iterative graph processing. Thus, the value of a subgraph at an iteration should be measured in terms of not only its UD but also its PUD.

Now, how do we extract the UD and PUD from a subgraph at a given iteration so that both can be transferred to GPU? Extracting the UD from a subgraph is easy as its active vertices in the current iteration are known (lines 4 and 12 in Figure 3). However, extracting precisely the PUD (without NUD) from a subgraph is difficult, as its future active vertices are not known yet during the current iteration.

For a given subgraph, we propose to predict its PUD size at an iteration from the UD sizes in the current and past iterations. This enables to adopt a value-driven differential scheduler that computes the value of a subgraph adaptively and schedules it depending on if it has a high value (when its UD and PUD are dominant) or a low value (otherwise).

3 Scaph Overview

Figure 7 shows the workflow of Scaph, in which all the subgraphs of a graph are computed on the GPU while the host is responsible for their preparation. At each iteration, its dispatcher classifies a subgraph into either a high-value or low-value subgraph and sends it to its corresponding engine to facilitate value-driven differential scheduling. Both engines schedule their subgraphs for acceleration on GPU independently but concurrently.

Value-Driven Subgraph Dispatcher. Conceptually, the value of a subgraph at a given iteration is proportional to the amount of its UD and PUD. The key insight here is that, for a given subgraph, although accurately computing its PUD is difficult, its PUD size can be approximated based on the UD sizes in the current and past iterations. For a subgraph at a given iteration, Scaph's subgraph dispatcher (§4), classifies it adaptively as a high-value subgraph if it contains a sufficient amount of UD and PUD to justify its transfer in its entirety to GPU and a low-value subgraph to request only its UD to be transferred to GPU otherwise. This is done adaptively as the value of a subgraph changes as the iteration progresses.

Value-Driven Subgraph Scheduler. Scaph has two separate graph processing engines, described in §5, to process differentially high- and low-value subgraphs. For a high-value subgraph, we use a queue-assisted multi-round processing engine, which streams it entirely from the host to GPU (if it is not in GPU memory) and exploits both its UD and PUD adequately to enable faster convergence. For a low-value subgraph, Scaph relies on the graph processing engine given in Figure 3 but transfers only its UD to GPU, with the UD extracted in a NUMA-aware manner on the host.

Scaph is essentially a hybrid graph system that allows out-of-order computation of high-value subgraphs in each synchronous iteration. The use of asynchronous execution allows fast convergence but also changes the vertex scheduling priority of subgraphs. Therefore, a graph algorithm can use Scaph safely for preserving the convergence and the converged values, if it satisfies the correctness condition that the final vertex results are insensitive to the value propagation order.

4 Value-Driven Subgraph Dispatching

In Section 4.1, we quantify the value of a subgraph. In Section 4.2, we discuss how to estimate the value of a subgraph to support value-driven differential scheduling.

4.1 Quantifying the Value of a Subgraph

Graph computations proceed iteratively until convergence. Conceptually, the value of a subgraph $\tilde{G}$ can be measured in terms of its UD used in the current iteration and its PUD used in future iterations. Therefore, the *value* of $\tilde{G}$, denoted $Val(\tilde{G})$, from the current iteration Cur to the MAX-th iteration (beyond which $\tilde{G}$ is no longer active), is defined as:

$$Val(\tilde{G}) = \sum_{i=Cur}^{MAX} \sum_{v \in \tilde{G}.SetOfVertices} D(v) * A^i(v) \tag{1}$$

where $D(v)$ represents the number of out-going edges of vertex v restricted to $\tilde{G}$ and $A^i(v) \in \{0,1\}$ indicates that v is active (inactive) in the i-th iteration when $A^i(v) = 1$ ($A^i(v) = 0$). $Val(\tilde{G})$ represents the amount of computations arising from $\tilde{G}$ from the current iteration until convergence. According to Equation (1), the PUD of a subgraph is quantized by the number of its edges that will be used in future iterations.

The value of a subgraph depends upon its active vertices and their degrees. In the case of uniform degree distributions, the activation status of vertices can still differentiate the amount of UD, PUD, and NUD for a subgraph.

4.2 Value-Driven Differential Scheduling

Scaph emphasizes value-driven data transfers, which should directly reflect how the bandwidth is *effectively* utilized in order to enable faster convergence.

The intuition behind $Val(\tilde{G})$ is clear. If $Val(\tilde{G})$ is high, $\tilde{G}$ should be a high-value subgraph. Then we should transfer $\tilde{G}$ as a whole to GPU and also exploit its UD and PUD adequately by iterating over $\tilde{G}$ multiple times before it is removed from GPU memory. Otherwise, $\tilde{G}$ should be treated as a low-value subgraph. In this case, we will opt to transfer only its UD to GPU and just iterate over the resulting $\tilde{G}$ once.

If $\tilde{G}$ is a high-value subgraph, then the throughput of processing $\tilde{G}$ on GPU can be measured as follows:

$$T_{HV}(\tilde{G}) \quad = \quad \frac{|UD| + \lambda |PUD|}{|\tilde{G}|/BW + t_{barrier}} \tag{2}$$

The denominator $|\tilde{G}|/BW + t_{barrier}$, which represents the data transfer time for $\tilde{G}$, is used to approximate the time elapsed on processing $\tilde{G}$ by assuming a complete overlap between data transfers and computations on GPU. As $\tilde{G}$ is transferred in its entirety to GPU, $|\tilde{G}|$ denotes the amount of data thus transferred, BW represents the host-GPU bandwidth, and $T_{barrier}$ is the synchronization overhead for $\tilde{G}$ (amortized by the number of active subgraphs processed). The numerator $|UD| + \lambda * |PUD|$ represents the amount of UD and PUD accessed when $\tilde{G}$ is iterated over on GPU. We use a balancing factor λ to decay $|PUD|$, where $0 \leqslant \lambda \leqslant 1$, to signify the actual amount of PUD accessed.

If $\tilde{G}$ is a low-value subgraph, then we have:

$$T_{LV}(\tilde{G}) \quad = \quad \frac{|UD|}{|UD|/BW + t_{barrier}} \tag{3}$$

This time, only the UD of $\tilde{G}$ is streamed to GPU.

Now, $\tilde{G}$ is a high-value subgraph if $T_{HV}(\tilde{G}) \geqslant T_{LV}(\tilde{G})$ and a low-value subgraph otherwise. Thus, we need to analyze:

$$|UD| + \lambda |PUD| (1 + \frac{t_{barrier}}{|UD|/BW}) > |\tilde{G}| \tag{4}$$

To verify $T_{HV}(\tilde{G}) \geqslant T_{LV}(\tilde{G})$, the key lies in determining $|PUD|$, which is difficult to obtain directly. In fact, for a subgraph, its PUD is technically activated from its UD, motivating us to estimate the PUD of a subgraph heuristically based on the UD of the same subgraph. In this work, we consider a subgraph to have a high value if either of the following two conditions (which we found to work well across all of our applications, as confirmed in §6) holds to simplify Equation (4):

```
1  Procedure VDDSEngine(Graph $G$)
2      Distribute $G$'s subgraphs $\{\tilde{G}_1,\cdots,\tilde{G}_n\}$ to NUMA nodes
3      VertexInitialization($G$)
4      $\tilde{G}_{active} \leftarrow$ FindActiveSubgraph($G$)
5      Transfer VertexStates from GPU to CPU
6      while $\tilde{G}_{active} \neq \emptyset$ do
7          foreach $\tilde{G} \in \tilde{G}_{active}$ do
8              if Predictor( $\tilde{G}$) = "HIGH-VALUE" then
9                  Push(HVworklist, $\tilde{G}$)
10             else
11                 Push(LVworklist, $\tilde{G}$)
12         HVSPEngine(HVworklist)
13         LVSPEngine(LVworklist, VertexStates)
14         $\tilde{G}_{active} \leftarrow$ FindActiveSubgraph($G$)
15         Transfer VertexStates from GPU to CPU
```

Figure 8: Value-driven differential scheduling for high- and low-value subgraphs, with the calls in blue executed on GPU

- $|UD|/|\tilde{G})| > \alpha$. This indicates that UD is dominant among $\tilde{G}$. Intuitively, $\tilde{G}$ is a high-value subgraph.
- $|UD_{current}| - |UD_{last}| > 0$ and $|UD|/|\tilde{G}| > \beta$. UD remains a medium level and is also growing increasingly over iteration, indicating the potentially growing PUD. $\tilde{G}$ can be thus treated as a high-value subgraph.

When α is relatively large, which implies that the UD in a subgraph tends to be dominant, we can determine if it is a high-value subgraph by considering only its UD. β is needed to identify the high-value subgraphs where the amount of UD is relatively low and that of PUD is potentially high. Thus, β is often smaller than α. As shown in Table 1, considering both together is often more effective than considering either alone. In this work, α and β are set empirically as 50% and 30% to represent a nice point for yielding good results.

Figure 8 gives our value-driven differential scheduler, `VDDSEngine()`, for scheduling a graph G. Initially, G is partitioned into subgraphs, $\tilde{G}_1, \cdots, \tilde{G}_n$, at the host and distributed across its NUMA nodes (to facilitate their scheduling). Scaph uses two graph processing engines, as described in §5 below, `HVSPEngine()` for scheduling high-value subgraphs, and `LVSPEngine()` for scheduling low-value subgraphs. In line 8, Scaph uses the above heuristic predictor to estimate the value of an active subgraph. Note that both engines work independently but concurrently. `LVSPEngine()` needs *VertexStates* in order to perform UD extraction for the active vertices in each subgraph. The UD extraction can be overlapped effectively with the data transfers in `HVSPEngine()`. At the end of each iteration (line 15), Scaph will transfer back the updated vertices from the GPU to the CPU. Edges, which are not modified, are thus not transferred.

5 Value-Driven Subgraph Processing

Scaph has two graph processing engines. We describe the one for handling high-value subgraphs in §5.1 and the one for handling low-value subgraphs in §5.2.

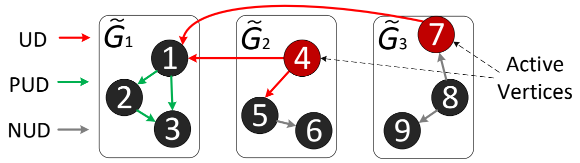


Figure 9: An example illustrating value propagation across the subgraphs, with ❶ activatable by ❹ and ❼. The PUD in $\tilde{G}_1$ can be exploited only if $\tilde{G}_2$ and/or $\tilde{G}_3$ are processed first.

5.1 High-Value Subgraph Processing

The key to extracting the most value out of high-value subgraphs lies in how to fully exploit their PUD. A useful idea of running each loaded subgraph multiple times is leveraged in the out-of-core settings [2, 53, 69] to exploit the *intrinsic value in a subgraph* for reducing the number of I/Os between memory and disk. However, under a GPU-accelerated heterogeneous architecture, subgraphs must often be small enough (in several tens of millions of bytes [17, 26]) against the ones in out-of-core settings, to enable fine-grained GPU scheduling. In this case, simply iterating over such a small-sized subgraph multiple times is often ineffective, since it can exploit only the PUD of its active vertices activated by its other active vertices but not active vertices from other subgraphs.

In Scaph, we improve the PUD exploitation significantly by enabling exploiting the *external value across the subgraphs*. Our key observation is that: *given a subgraph already available in GPU memory, scheduling it again* ***after a period of delay*** *can expose its PUD more fully than processing it repeatedly*. Figure 9 illustrates this with three subgraphs, exhibiting some complex inter-subgraph data dependencies (as is often the case in practice). We see that ❶ in $\tilde{G}_1$ can be activated by ❹ in $\tilde{G}_2$ and ❼ in $\tilde{G}_3$. Once ❶ in $\tilde{G}_1$ is activated, ❷ in $\tilde{G}_1$ may get activated (as shown). In this case, the edge data for ❶→❷, ❶→❸, and ❷→❸ are part of the PUD of $\tilde{G}_1$. By processing $\tilde{G}_1$ after $\tilde{G}_2$ or $\tilde{G}_3$ or both (even better), we can exploit such PUD to enable faster convergence. That is, repeatedly processing $\tilde{G}_1$ would not help.

Queue-Assisted Multi-Round Processing. The scheduling of high-value subgraphs at a given iteration is shown in Figure 10. We use a k-level priority queue ($PQ_1, \ldots, PQ_k$) to enable re-scheduling a GPU-resident subgraph after some delay, where k indicates the maximum number of times some subgraphs have been processed in the current iteration. Thus, k varies from iteration to iteration. Figure 11 shows a case.

In each differential scheduling iteration orchestrated by `VDDSEngine` (Figure 8), `HVSPEngine(worklist)` is invoked, where *worklist* contains all the high-value subgraphs in this iteration. During the pre-processing (lines 2–6), each subgraph $\tilde{G}_i$ in *worklist* is examined in turn. $\tilde{G}_i$ will be enqueued into PQ_1 (if not already there) if $\tilde{G}_i$ remains to be GPU-resident (i.e., in one of $\{PQ_1, \ldots, PQ_k\}$) from the previous iteration and inserted into *TransSet* (waiting to be streamed to GPU) otherwise. Thus, there are two concurrently executed modules,

```
1  Procedure HVSPEngine(worklist)
2      foreach $\tilde{G}_i \in$ worklist do
3          if $\tilde{G}_i$ is resident in GPU memory then
4              Push($PQ_1$, $i$)
5          else
6              Push(TransSet, $i$)
       /* Subgraph Transferring Module */
7      while $TransSet \neq \emptyset$ do
8          if copystream is available then
9              $i \leftarrow$ Pop(TransSet)
10             if GPU has available memory for one subgraph then
11                 $Gbuf \leftarrow$ AllocateDeviceMemory( )
12             else
13                 $j \leftarrow$ Pop($PQ_k$)
14                 $Gbuf \leftarrow$ GetGbuf($\tilde{G}_j$)
15             TransferData(copystream, Gbuf, $\tilde{G}_i$, CPU2GPU)
16             Push($PQ_1$, $i$)
       /* Subgraph Scheduling Module */
17     while $worklist \neq \emptyset$ do
18         if at least one stream in execstreams is available then
19             stream $\leftarrow$ Available(execstreams)
               /* Exploit the UD of a subgraph in $PQ_1$ */
20             if $PQ_1 \neq \emptyset$ then
21                 $i \leftarrow$ Pop($PQ_1$)
22                 Kernel(stream,$\tilde{G}_i$)
23                 Erase(worklist, $\tilde{G}_i$)
               /* Exploit the PUD of a subgraph in $PQ_i$, where $i \neq 1$ */
24             else
25                 for $p \leftarrow 2$ to $k$ do
26                     if $PQ_p \neq \emptyset$ then
27                         $i \leftarrow$ Pop($PQ_p$)
28                         Kernel(stream,$\tilde{G}_i$)
29                         break
30             priority $\leftarrow$ GetPriority($\tilde{G}_i$)
31             Push($PQ_{priority+1}$, $\tilde{G}_i$)
```

Figure 10: High-value subgraph processing in each iteration (called from Figure 8). The Kernel function is from Figure 3. The two colored code regions are executed in parallel.

Subgraph Transferring and Subgraph Scheduling.

The Subgraph Transferring module (lines 7 – 16) is responsible for streaming asynchronously the subgraphs in *TransSet* to GPU. This is done by using some free GPU memory whenever possible (line 11) or making some free by dequeuing a subgraph from the multi-level queue (lines 13 – 14). Due to lines 4 and 16, all subgraphs in *worklist* are initially enqueued into PQ_1, and thus assigned with the highest priority.

The Subgraph Scheduling module (lines 17 – 31) is responsible for scheduling the subgraphs in $PQ_1, \cdots, PQ_k$. The subgraphs in PQ_1 are processed first (for the first time in the current iteration) to exploit their UD (lines 21 – 23). If $PQ_1 = \emptyset$ (implying that some subgraphs are still being transferred to GPU asynchronously), the scheduler will dequeue a subgraph from a non-empty PQ_i, where i is the smallest, to exploit its PUD (lines 25 – 29), as this will be the i-th time that the subgraph is processed (in the i-th round) of the current iteration. Simultaneously, the data transfers for PQ_1 and the computations for $PQ_2, \cdots, PQ_k$ are maximally overlapped, too. In either case, the priority of a subgraph, once processed, drops by one (lines 30 – 31). This delayed re-scheduling attempts to maximize the PUD exploitation, by, e.g., increasing the chances for $\tilde{G}_1$ to be processed after $\tilde{G}_2$ and/or $\tilde{G}_3$ in Figure 9 (as motivated earlier). Consider $\tilde{G}_3$, which resides in PQ_1, in Figure 11. Once we have exploited its UD, we will move it to PQ_2 so that we can exploit its PUD after $\tilde{G}_7$, $\tilde{G}_4$, $\tilde{G}_8$, and $\tilde{G}_6$ have been processed.

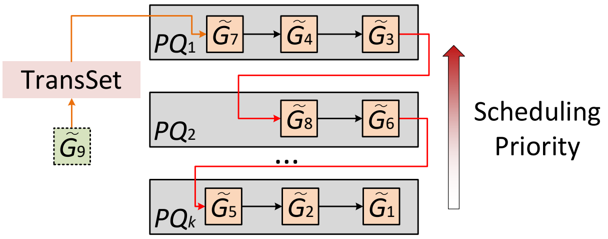


Figure 11: Subgraph processing with a k-level priority queue. PQ_i represents a queue PQ with the i-th priority. The smaller i is, the higher the priority is. All the subgraphs streamed from the host to GPU enter into PQ_1 initially.

Our scheduler with a multi-level priority queue guarantees that subgraphs are scheduled fairly, preventing them from bearing too many *useless computations* in the sense that the data of a vertex is computed but not updated.

Time and Space Complexity Analysis. k is expected to be bounded by $\frac{BW'}{BW}$ where BW' is the internal bandwidth of GPU and BW is the host-GPU bandwidth. In our computing platform, $BW' = 224$GB/s and $BW = 11.4$GB/s. Thus, $k \leqslant 20$ is typically expected.

As for the space complexity, a k-level priority queue is used to keep track of only the indices of the active subgraphs processed in an iteration. Thus, the worst complexity is $O(\frac{Mem_{GPU} \times \text{sizeof(SubgraphIndex)}}{|\tilde{G}|})$, where Mem_{GPU} is the global memory size and $|\tilde{G}|$ is the size of a subgraph $\tilde{G}$. In our computing platform, we have used $\frac{4\text{GB} \times 4\text{B}}{32\text{MB}} = 0.5$KB.

5.2 Low-Value Subgraph Processing

The key to exploiting the most value of low-value subgraphs is to extract their UD efficiently. We use multiple CPU cores at the host to parallelize the UD extraction. Due to *non-uniform memory access* (NUMA), however, scanning naively all the vertices in a subgraph to extract its UD can still be costly. In addition, different subgraphs exhibit different amounts of UD. Such scanning tasks are also prone to load imbalance.

Figure 12 gives our scheduler for low-value subgraphs. In each value-driven scheduling iteration orchestrated by `VDDSEngine()` in Figure 8, `LVSPEngine(worklist, VertexStates)` is invoked, where *worklist* contains all the low-value subgraphs that are active in this iteration, with their active vertices indicated in *VertexStates*. There are three modules, UD Extraction, Subgraph Transferring, and Subgraph Scheduling, which all execute concurrently. The major contribution here is a NUMA-aware parallel UD extraction.

UD Extraction. Initially, all the subgraphs partitioned from a graph are evenly distributed to different NUMA nodes,

```
1  Procedure LVSPEngine(worklist, VertexStates)
       /* UD Extraction Module */
2      para_for $\tilde{G}_i \in$ worklist do
3          $\tilde{G}'_i \leftarrow$ UDExtraction($\tilde{G}_i$, VertexStates)
4          Push(TransSet, i )
       /* Subgraph Transferring Module */
5      while TransSet $\neq \emptyset$ do
6          if copystream is available then
7              $i \leftarrow$ Pop(TransSet)
8              Gbuf $\leftarrow$ AllocateDeviceMemory()
9              TransferData(copystream, Gbuf, $\tilde{G}'_i$, CPU2GPU)
10             Push(ExecFIFO, i )
       /* Subgraph Scheduling Module */
11     while worklist $\neq \emptyset$ do
12         if at least one stream in execstreams is available then
13             stream $\leftarrow$ Available(execstreams)
14             $i \leftarrow$ Pop(ExecFIFO)
15             Kernel(stream, $\tilde{G}'_i$)
16             Erase(worklist, $\tilde{G}_i$)
   /* Extract UD on the host */
17 Procedure UDExtraction($\tilde{G}$, VertexStates)
18     $\tilde{G}' \leftarrow \emptyset$
19     Offset $\leftarrow 0$
20     while offset $\leqslant |\tilde{G}$.SetOfVertices$|$ do
21         WORD $\leftarrow$ Load(VertexStates($\tilde{G}$).bitmap, offset, 32)
22         if WORD $\neq 0$ then
23             foreach BYTE $\in$ WORD do
24                 if BYTE $\neq 0$ then
25                     foreach BIT $\in$ BYTE do
26                         if BIT = 1 then
27                             $v \leftarrow$ GetVert(offset, BYTE,
                                   BIT)
28                             $\tilde{G}' \leftarrow \tilde{G}' \bigcup v$.outedges
29         offset $\leftarrow$ offset + 32
30     return $\tilde{G}'$
```

Figure 12: Low-value subgraph processing in each iteration (called from Figure 8). The Kernel function is from Figure 3. The three shaded code regions are executed in parallel.

with a NUMA node consisting of a CPU socket and its own memory banks (line 2 in Figure 8). The UD extraction module is given in terms of lines 2 – 4 and lines 17 – 30. To boost performance and improve intra-node load balancing, the UD extraction for each subgraph is done in its own thread, which is bound to the NUMA node storing the subgraph (line 3). To improve inter-node load balancing (as a minor optimization), we also duplicate in a NUMA node an equal number of randomly selected subgraphs from the other nodes (if there is still some memory space available). We adopt a simple bitmap-based approach to extract the UD from a subgraph $\tilde{G}$ efficiently (lines 17–30). All its vertices are stored in a bitmap, $VertexStates(\tilde{G}).bitmap$, with 1 (0) indicating that the corresponding vertex in $\tilde{G}$ is active (inactive). To accelerate its construction, the total of active vertices is computed on GPU.

Unlike high-value subgraphs, which can each be stored in the same-sized chunk in GPU memory (§5.1), low-value subgraphs may give rise to UD-induced subgraphs of varying sizes. To reduce fragmentation, Scaph further divides each chunk for storing a subgraph into smaller tiles (totaling 32 in our implementation). To store a UD-induced subgraph in GPU memory, Scaph will try to find consecutive tiles first in a partially filled chunk and then in a vacant chunk.

Subgraph Transferring. As in the case of high-value subgraph streaming in Figure 10, this module proceeds similarly except that a multi-level queue is no longer used.

Subgraph Scheduling. As in the case of scheduling high-value subgraphs to GPU in Figure 10, this module schedules UD-induced subgraphs (without using a multi-level queue).

6 Evaluation

We evaluate the efficiency and scalability of Scaph by answering the following four research questions (RQs):

- ***RQ1:*** How much more efficient is Scaph over state-of-the-art heterogeneous graph systems?
- ***RQ2:*** How effective is Scaph's value-driven differential scheduling in helping it achieve the overall performance?
- ***RQ3:*** How well does Scaph scale?
- ***RQ4:*** How much runtime overhead does Scaph introduce?

6.1 Experimental Setup

We compare Scaph with the following three state-of-the-art CPU-GPU heterogeneous graph systems:

- Totem [14]. A graph is divided into two subgraphs, which are processed by CPU and GPU, respectively. At the end of each iteration, the states of the active vertices that are activated reciprocally by the two subgraphs are exchanged.
- *Graphie* [17]. Like Scaph, a graph is initially partitioned at the host CPU and the subgraphs are then streamed to GPU for graph processing. Unlike Scaph, however, all active subgraphs are transferred to GPU in their entirety.
- *Garaph* [34]. At an iteration, all the subgraphs that are partitioned from a graph are processed concurrently by both the host and GPU if the number of outgoing edges of all active vertices in the entire graph exceeds 50% of the total number of edges and on the host only otherwise.

Subgraph Size. For Totem, Graphie, and Garaph, the sizes of subgraphs are selected from their papers. In Scaph, a graph is partitioned into subgraphs of 32MB each for several reasons. First, the host-GPU bandwidth tends to be under utilized with smaller sizes. Second, subgraphs will be streamed to GPU more frequently with larger sizes, as they tend to contain active vertices for more iterations. Finally, the kernel launching overheads appear to be well hidden with 32MB.

Graph Applications. We consider the first three typical graph algorithms (from different categories) and the latter two actual graph workloads (with different complexities): (1) *Single-Source Shortest Path* (SSSP) [60]–Sequential traversal, (2) *Connected Components* (CC) [20]–Parallel traversal, (3) *Minimum Spanning Tree* (MST) [37]–Graph mutation, (4) *Neural Network Digit Recognition* (NNDR) [4], and (5) *Graph-based Circuit Simulation* (GCS) [25]. All these algorithms fit the correctness criteria discussed in §3, though NNDR and GCS are already typically executed in an asyn-

Table 2: Graph datasets. The graph size is evaluated in the weighted edgelist representation.

Dataset	#Vertices	#Edges	Avg. Degree	Size
twitter (TW)	41.7M	1.47B	39.5	32.8GB
comfriend (FR)	124.8M	1.81B	14.5	40.4GB
sk-2005 (SK)	50.6 M	1.95B	38.5	43.6GB
uk-2007 (UK)	105.9M	3.74B	35.3	83.6GB
altavista-2002 (AV)	1.41G	6.64B	4.695	148.3GB
fb-2009 (FB)	139.1M	12.3B	88.7	275.6GB
RMAT-k (25<k<31)	2^k	2^{k+4}	16	-

Table 3: Execution times of Scaph, Totem, Graphie, and Garaph. Here, 'N/A' indicates that a graph algorithm has abnormally terminated due to some runtime error.

Algorithm	System	Execution Time (Secs)					
		TW	FR	SK	UK	AV	FB
CC	Totem	2.41	5.01	2.72	9.32	N/A	N/A
	Graphie	1.89	4.46	16.53	23.61	57.49	133.21
	Garaph	1.17	2.53	2.90	7.07	31.46	86.24
	Scaph	0.28	0.91	1.08	2.47	7.08	15.35
SSSP	Totem	5.94	5.78	7.07	19.21	N/A	N/A
	Graphie	5.32	9.24	52.01	89.44	218.51	413.07
	Garaph	3.71	4.45	6.83	16.52	114.68	204.35
	Scaph	0.92	1.67	3.17	6.64	29.06	38.87
MST	Totem	7.93	10.90	21.33	42.84	N/A	N/A
	Graphie	8.45	16.24	32.19	53.22	198.85	304.51
	Garaph	4.14	7.38	12.35	25.82	101.25	131.45
	Scaph	1.39	1.99	2.93	6.36	25.23	35.41
NNDR	Totem	6.47	6.63	12.17	29.43	N/A	N/A
	Graphie	5.38	7.32	28.19	49.81	234.04	457.13
	Garaph	3.41	4.76	9.28	28.74	116.34	175.34
	Scaph	1.77	2.08	2.99	5.13	20.19	33.55
GCS	Totem	19.77	23.04	59.51	98.11	N/A	N/A
	Graphie	24.08	38.84	50.34	93.29	454.41	834.59
	Garaph	10.53	15.56	20.438	39.45	185.58	299.76
	Scaph	3.33	4.08	10.46	16.13	39.52	54.94

chronous way, while the other algorithms are typically run in a synchronous, iterative manner.

Graph Datasets. We use (1) 6 real-world graphs [6, 31]) for performance evaluation, and (2) 5 large synthesized graphs (generated by the RMAT tool [7]) for scalability evaluation. Table 2 gives all the graphs used. For SSSP and MST that work on the weighted graphs, we randomly assign each edge of an unweighted graph with a weight ranging from 1 to 100.

Computing Platform. We evaluate Scaph on a machine where the host is equipped with two Intel 14-core Xeon CPUs, E5-2680v4@2.40GHz with 512GB memory (256GB on each of the two NUMA nodes). The GPU is NVIDIA P100 (with 56 SMXs, 3584 cores, and 16GB memory), connected to the host via the PCI Express 3.0 at 16x. The host-GPU bandwidth is around 11.4GB/s. We use NVCC V8.0.61 and g++ V5.4.0 to compile all the applications under "-O3". The operating system is Ubuntu 14.04 with Linux kernel 4.13.

6.2 RQ1: Efficiency

To answer RQ1, we compare Scaph against Totem [14], Graphie [17], and Garaph [34]. Table 3 depicts the results.

Scaph vs. Totem. The speedup of Scaph over Totem ranges from 2.23× (for SSSP on SK) to 7.64× (for CC on TW) with an average of 4.12×. Totem's critical performance bottleneck lies in its severe load imbalance, as it partitions each graph into only two subgraphs, one for the host (with 512GB memory) and one for GPU (with only 16GB memory). As a result, Totem cannot tap GPU's processing power to exploit adequately the UD and PUD in a graph. Its bottleneck is to ask the CPU to process most of the graph data, which would have been processed more efficiently by the GPU otherwise. A typical measurement for FB is for the CPU to handle 358.1GB and the GPU to handle only 16GB. In contrast, Scaph streams all subgraphs dynamically to GPU with value-driven differential scheduling, thereby exploiting more adequately GPU's processing power, and consequently, the UD and PUD in all the subgraphs. In the case of CC operating on FR, SK, and UK, their GPU portions under Totem are 39.6%, 36.7%, and 19.1%, respectively. As a result, Scaph outperforms Totem by 5.51x (FR), 2.52x (SK), and 3.77x (UK).

Scaph vs. Graphie. Scaph is faster than Graphie by 8.93× on average, with its speedup ranging from 3.03× (for NNDR on TW) to 16.41× (for SSSP on SK). Both Graphie and Scaph process all subgraphs on GPU only. So Graphie can be understood as a version of Scaph, where every subgraph is treated as a high-value subgraph except that only its UD is used but its PUD is exploited rather inadequately. Graphie is inferior to Scaph for several reasons. First, Graphie transfers an active subgraph entirely to GPU even though it contains only a few active vertices (i.e., a lot of NUD), wasting the host-GPU bandwidth. Second, Graphie exploits the UD only but PUD inadequately in an active subgraph.

Let us examine SSSP on SK, where the speedup of Scaph over Graphie is the highest (at 16.41×). Graphie converges in 75 iterations, by transferring 18,019 subgraphs totaling 374.4GB data to GPU. In contrast, Scaph converges in 16 iterations, by transferring 9,897 subgraphs totaling only 19.6GB data, comprising 13.2GB for 798 high-value subgraphs and 6.4GB for 9,099 low-value subgraphs. For Scaph, its significantly improved utilization for the host-GPU bandwidth has resulted in its significantly improved overall performance.

Scaph vs. Garaph. Scaph is faster than Garaph by 3.71×, with an overall rang from 1.93× (for NNDR on TW) to 5.62× (for CC on FB). Unlike Scaph, Garaph processes all the subgraphs on both the host and GPU if the active vertices in the entire graph have a lot of outgoing edges and on the host only otherwise (§6.1). Despite this, Garaph cannot distinguish high-value from low-value subgraphs as Scaph does. While being more effective than Graphie in reducing the amount of NUD transferred, Garaph is inferior to Scaph as it still transfers more NUD to GPU and exploits PUD less adequately.

Let us examine CC on FB, where the speedup of Scaph over Garaph is the highest (at 5.62×). Garaph processes all the subgraphs on the host only (as the outgoing edges of FB's active vertices over the total is under 6.9% at any iteration), by

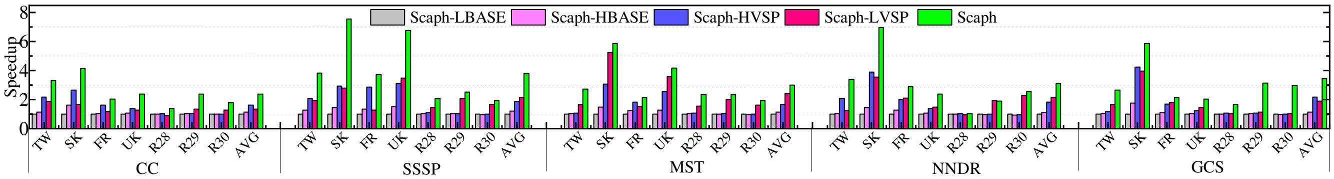


Figure 13: Speedup of Scaph, Scaph-HVSP, and Scaph-LVSP (normalized to Scaph-LBASE)

using a so-called notify-pull model. In contrast, Scaph uses a fine-grained value-driven differential scheduler to identify high-value and low-value subgraphs even though it has active vertices only in its local regions at any iteration, so that the GPU's processing power is adequately exploited.

6.3 RQ2: Effectiveness

To answer RQ2, we consider four variations of Scaph: (1) **Scaph-HVSP**, where all the low-value subgraphs can be understood as being misidentified as high-value subgraphs, (2) **Scaph-LVSP**, where all the high-value subgraphs can be understood as being misidentified as low-value subgraphs, (3) **Scaph-HBASE**, which applies the differential processing, but every subgraph transferred to the GPU has kept computation with a specific number of times (without using queue-based delayed scheduling), as used in CLIP [2], and (4) **Scaph-LBASE**, a variation of Scaph-LVSP except that every subgraph is streamed to GPU entirely (without UD extraction), as used in Graphie [17].

Figure 13 gives the results. We see that neither of Scaph-HVSP and Scaph-LVSP is always better than the other, and also Scaph is the best performer for all the algorithms on all the graphs. Thus, Scaph's value-driven differential scheduling with heuristic subgraph identification is highly effective.

Scaph-HVSP. Scaph-HVSP achieves better speedups for the graphs where algorithms take longer iterations to converge, as this allows it to exploit PUD more adequately and thus stream less redundant data to GPU. For example, each algorithm on SK has the longest number of iterations against on other graphs, thereby delivering considerable speedups. We also see that Scaph-HBASE is significantly inferior to Scaph-HVSP. This is because small subgraphs often contain very little PUD from themselves worthy of being exploited. Our queue-based scheduling allows the availability of PUD from other subgraphs via delayed scheduling. Thus, multi-time processing under Scaph-HVSP can expose significantly more PUD than that under Scaph-HBASE (i.e., by simply applying the idea from CLIP) for boosting performance.

Scaph-LVSP. Just like Scaph-HVSP, Scaph-LVSP can be quite effective in some cases. For example, the top two speedups achieved by Scaph-LVSP for MST are 5.26x and 3.58x on SK (14.8GB) and UK (27.61GB), respectively. The corresponding speedups from Scaph are 5.99x and 4.19x. However, Scaph-LVSP can be rather ineffective for the graphs that can nearly fit into the 16GB GPU memory, since Scaph-LBASE will then make GPU-resident for nearly all the subgraphs. For R28 with 16.78GB (unweighted) and 29.48GB (weighted), Scaph-LVSP offers little or even negative benefits for CC, NNDR, and GCS (on unweighted graphs) but positive ones for SSSP and MST (on weighted graphs).

Scaph. Scaph obtains the best of both worlds, Scaph-HVSP and Scaph-LVSP. For CC, SSSP, MST, NNDR, and GCS, the average speedups achieved by Scaph-HVSP (Scaph-LVSP) are 1.63x, 1.87x, 1.66x, 1.84, and 2.18x (1.33x, 2.12x, 2.41x, 2.15x, and 1.90x), respectively. As for Scaph, these average speedups are 2.38x, 3.79x, 3.01x, 3.12x, and 3.44x. Note that Scaph has the highest gain on SK, where Scaph-HVLP and Scaph-LVSP are also most effective.

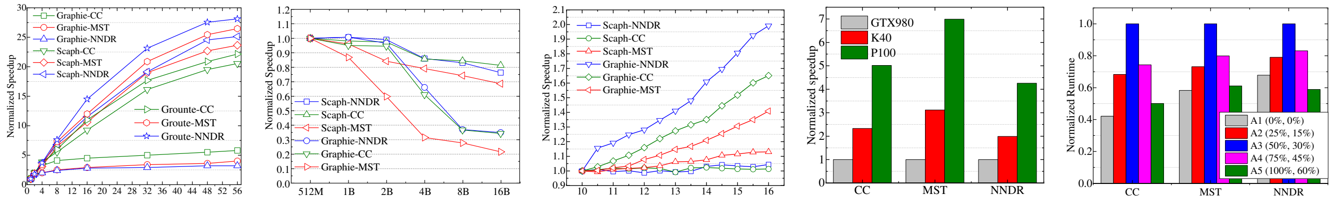
6.4 RQ3: Sensitivity Study

To answer RQ3, we investigate Scaph's scalability in terms of #SMXs, graph sizes, memory sizes, and GPU generations. We select Graphie as a reference on CC, MST, and NNDR.

#SMXs. Figure 14(a) compares Scaph and Graphie in terms of CC, MST, and NNDR on UK [6] for varying #SMXs by using all the 8GB GPU memory available. Scaph is significantly more scalable than Graphie for all the three graph algorithms, since Scaph can utilize the host-GPU bandwidth more effectively as already motivated earlier (Figures 1 and 4). For example, Graphie-MST reaches its plateau when #SMXs = 2, but Scaph-MST continues to offer a scalable performance improvement. CC and NNDR exhibit a similar trend.

However, Scaph's scalability degrades gradually as #SMXs increases, due to the integrated impacts of the intrinsic random accesses of graph processing on GPU [5, 25, 57] and the increasingly more SMXs competing for the memory bandwidth. As also shown in Figure 14(a), Groute [5], an in-memory graph system that can not handle over-subscription, on UK-2007@1M [6] (a sample graph with 1M vertices and 41M edges generated from UK), suffers from exactly the same scalability problem, which is beyond the scope of this work. We leave addressing this problem in future work.

Graph Sizes. Figure 14(b) compares Scaph and Graphie as the graph size increases. For CC and NNDR working on unweighted graphs, Scaph (Graphie) can store up to 4 billion (2 billion) edges in GPU memory. For MST working on the weighted graph, these edge counts drop to roughly 2 billion and 1 billion. Both Scaph and Graphie maintain their throughput well as the graph size increases but degrade visibly for the graphs that can no longer fit into GPU memory. However,



(a) #SMXs (b) Edge sizes (c) GPU memory (GB) (d) GPU generations (e) Varying α and β

Figure 14: Performance of Scaph and Graphie (including Groute for (a) only) in terms of varying (a) #SMXs: UK [6] and 8GB GPU memory, (b) graph sizes: R26–R30 [7] and 16GB GPU memory and #SMX=56, (c) GPU memory capacities: FB [31] and #SMX=56, (d) GPU genreations: FB [31], and (e) configurations of (α, β): FB [31], respectively. All results are normalized to the one obtained by itself with the smallest configuration.

Scaph has a slower performance reduction rate than Graphie, for two reasons. First, Scaph can better tap GPU's processing power due to its use of a multi-level priority queue for exploiting PUD more adequately and overlapping data transfers and GPU computation more effectively. Second, Scaph avoids transferring a large amount of NUD for low-value subgraphs.

GPU Memory Capacities. Figure 14(c) compares Scaph and Graphie for varying GPU memory sizes. Graphie is highly sensitive to the GPU memory capacity used, which determines directly how many subgraphs can be resident on GPU at an iteration and how many of these get re-processed in the ensuing iteration (before they are removed from GPU memory). In contrast, Scaph is nearly insensitive, since it exploits UD and PUD for high-value subgraphs and UD only for low-value subgraphs always. Note that Scaph is significantly faster than Graphie (Table 3). In Figure 14(c), Graphie improves over itself (normalized to 10GB) as the GPU memory size increases.

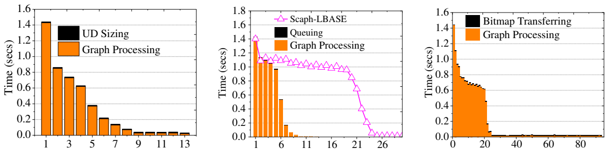
GPU Generations. Figure 14(d) characterizes the performance of Scaph on different GPU generations. Compared to Graphie that shows few speedups as shown in Figure 1, Scaph enables the significant speedups for K40 (1.99×∼3.12×) and P100 (4.26×∼5.02×) against that of GTX980.

Varying α and β. Figure 14(e) shows the sensitivity of the performance results of Scaph with respect to α and β. Here, A1 can be understood as Scaph-HVSP and A5 as Scaph-LVSP. Looking at A3, we see that increasing α and β causes more subgraphs to be mis-identified as low-value subgraphs (A4 and A5) and decreasing α and β causes more subgraphs to be mis-identified as high-value subgraphs (A1 and A2). Thus, A3 seems to represent a nice sweet spot for yielding good performance results. As for the problem of finding an optimal setting, we leave it as future work.

6.5 RQ4: Runtime Overhead

We discuss Scaph's overheads incurred in its *value-driven differential scheduling* (VDDS) given in Figure 8, *high-value subgraph processing* (HVSP) given in Figure 10, and *low-value subgraph processing* (LVSP) given in Figure 12.

VDDS. The cost of computing the subgraph value comes from computing the UD size for each iteration, on GPU, in line 8 of Figure 8. This is negligible, as shown in Figure 15(a).



(a) VDDS (b) HVSP (c) LVSP

Figure 15: Scaph's runtime overhead for running CC on UK [6] across the iterations

HVSP. The main overhead of HVSP lies in its queue management. In Figure 15(b), the cost incurred per iteration is small, representing an average of 0.79% of the total processing time. This small overhead is more than offset by the benefit reaped. In particular, the iteration count is reduced since most of the PUD can be computed ahead of schedule. The per-iteration time can be improved mainly because most of the NUD is discarded (rather than transferred expensively).

LVSP. The main overhead of LVSP lies in transfering a bitmap representation for all the active vertices in a subgraph from GPU to the host. As shown in Figure 15(c), the average cost incurred per iteration represents 4.3% of the total graph processing time. However, this cost increases relatively towards the last few iterations, reaching 57.4% at the end, where each subgraph has little UD to be acted upon.

6.6 Limitations

Graph Partition. Various partitions may show different value variations of subgraph at runtime. Scaph adopts a greedy vertex-cut partition [15] with the time taken depending on the number of partitions. It would be interesting future work to find a more reasonable partition method that can make most of UD and PUD exploited in the early stage of graph processing for faster convergence.

Disk-based Heterogeneous Graph Systems. The performance of Scaph is insensitive to the difference between CPU and GPU memory, given that the whole graph is assumed to fit into the CPU memory. To support even larger graphs on a single machine, using the disk (e.g., SSD) as secondary storage is promising. In this case, a new dimension of performance bottleneck will be the I/O inefficiency, which has been studied in prior work [2, 32, 35, 55]. We can combine Scaph with these past disk-based solutions to cooperatively handle graphs that cannot fit into the host memory.

Performance Profitability. Scaph delivers performance benefits by processing all the subgraphs differentially. Scaph is currently not expected to be applied to graph algorithms where the set of active vertices does not shrink as computation goes on. For example, all vertices in PageRank are active in every iteration. Thus, all the data of a subgraph can be regarded as UD without any PUD. In fact, we can extend Scaph to distinguish these all-active subgraphs further for PageRank by considering not only the degrees and the activation but also the state variation rate for each vertex, which is a potential direction of future work.

7 Related Work

Heterogeneous Graph Systems. Such systems have been studied on a range of heterogeneous architectures equipped with varying hardware resources [21, 35, 44]. Compared to GPU-accelerated solutions [43, 47], FPGA-accelerated alternatives are advantageous in energy-efficiency [10, 61]. In developing Scaph, we focus on improving host-accelerator bandwidth utilization. The basic idea behind can also be applied to improve the scalability of FPGA-accelerated heterogeneous graph systems with a few hardware specializations.

Distributed Graph Systems. The rationale is to aggregate multiple machines to enable processing large-scale graphs. The main challenge lies in obtaining good graph partitions [3, 8, 16, 48, 52] so as to minimize the communication overheads across the machines. Some recent studies take advantage of emerging high-speed networks (e.g., RDMA) to reduce communication overheads [49, 58]. Aspire [54] designs a relaxed consistency model to exploit asynchronous parallelism for iterative algorithms. Gemini [67] includes a series of adaptive runtime optimizations to enable obtaining an attractive scale-out efficiency.

Disk-based Graph Systems. Many disk-based graph systems [12, 45, 64, 65] exist for supporting large-scale graph processing. GraphChi [30] relies on parallel sliding windows to optimize disk accesses. GridGraph [68] uses 2-level hierarchical partitioning to reduce the I/O overhead. TurboGraph [18] applies a pin-and-slide model to exploit the multicore and I/O parallelism. Due to the low disk-to-memory bandwidth, disk-based graph systems are often at least two orders-of-magnitude slower than heterogeneous solutions.

Data Movement Reduction. Several previous studies leverage an analogous idea of running graph partitions multiple times for different purposes. CLIP [2] iterates over each loaded subgraph multiple times to squeeze out the value of each subgraph so that less amount of disk I/O is required. GraphQ [69] enables computing the local subgraphs multiple times in order to tolerate long latency across the compute nodes. Unlike these efforts, Scaph emphasizes on a GPU context that often requires small-size subgraphs to enable fine-grained scheduling. Thus, simply computing a subgraph multiple times is not sufficient to exploit its PUD fully. Scaph enables value exploitation not only within a subgraph but also *across the subgraphs* via a delayed scheduling mechanism.

In LUMOS [53], a subgraph in an iteration can be exploited asynchronously iff its updated values are independent of the subsequent iteration. This dependency-aware technique allows enjoying the efficiency of asynchronous execution while ensuring synchronous processing semantics. Applying this technique into Scaph can help identify the high-value subgraphs that contain across-iteration dependencies, so that Scaph can be extended to handle synchronous algorithms [22] safely by scheduling these high-value subgraphs once. However, the downside is that many dependency-free low-value subgraphs may also be allowed to be computed multiple times, wasting the GPU computational and storage resources.

Wonderland [63] uses graph abstraction as a bridge over on-disk subgraphs to speed up convergence. However, under the context of small-sized subgraphs, such a graph abstraction is often hard to keep concise, and extracting it from the whole graph is also non-trivial. PowerLayer [8] presents differentiated processing on high-degree and low-degree vertices to improve the trade-off between load balance and communication overheads in a distributed setting. However, applying the idea of PowerLayer cannot often identify the value of a subgraph accurately while Scaph does with a fine-grained solution. Mosaic [35] adopts a subgraph compression technique, which can be used to work together with Scaph to improve the bandwidth-efficiency of heterogeneous graph system further.

8 Conclusion

This paper tackles the challenge faced in achieving scale-up large-scale graph processing on a GPU-accelerated heterogeneous architecture. We introduce Scaph, a value-driven heterogeneous graph system that differentially schedules the subgraphs partitioned from a graph according to their values in order to improve the effective utilization of the host-GPU bandwidth. Scaph outperforms state of the art, as evaluated with representative graph algorithms operating on a range of graph datasets. In addition, these performance benefits scale up as more computing resources are available.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. In particular, we thank our shepherd, Xiaosong Ma, for her valuable suggestions. We would also like to thank Pengcheng Yao, Chuangyi Gui, Qinggang Wang, and Jieshao Zhao for their support. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, National Natural Science Foundation of China under Grant No. 61702201, 61825202, 61832006 and 61929103, and Australian Research Council DP180104069. The correspondence of this paper should be addressed to Xiaofei Liao.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117. IEEE, 2015.

[2] Zhiyuan Ai, Mingxing Zhang, and Yongwei Wu. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 125–137, 2017.

[3] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. In *Proceedings of the Hadoop Summit*, volume 11, pages 5–9, 2011.

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Proceedingns of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 235–248. ACM, 2017.

[6] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, pages 595–601, Manhattan, USA, 2004. ACM.

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*, pages 442–446. SIAM, 2004.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems (Eurosys)*, pages 13–21. ACM, 2015.

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[10] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 217–226. ACM, 2017.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (USENIX FAST)*, pages 309–316. USENIX, 2019.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 495–510. ACM, 2017.

[14] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–354. ACM, 2012.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30. USENIX, 2012.

[16] Joseph E. Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[17] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

[18] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 77–85. ACM, 2013.

[19] Pawan Harish and Petter J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 2007 International Conference on high-performance computing (HiPC)*, pages 197–208. Springer, 2007.

[20] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70(6):25–43, 2017.

[21] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88. IEEE, 2011.

[22] Unit Kang, Duen Horng "Polo" Chau, and Christos Faloutsos. Inference of beliefs on billion-scale graphs. In *Proceedings of KDD Workshop on Large-scale Data Mining: Theory and Applications (LDMTA)*, pages 1–7, 2010.

[23] Gary J. Katz and Joseph T. Kider. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 47–55. Eurographics Association, 2008.

[24] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (Eurosys)*, pages 169–182. ACM, 2013.

[25] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High Performance Parallel and Distributed Computing (HPDC)*, pages 239–252, 2014.

[26] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 447–461. ACM, 2016.

[27] Efstathios Kirkos, Charalambos Spathis, and Yannis Manolopoulos. Data mining techniques for the detection of fraudulent financial statements. *Expert systems with applications*, 32(4):995–1003, 2007.

[28] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Gpu-accelerated graph clustering via parallel label propagation. In *Proceedings of ACM Conference on Information and Knowledge Management (CIKM)*, page 567–576, 2017.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600. ACM, 2010.

[30] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46. USENIX, 2012.

[31] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[32] Hang Liu and Howie H. Huang. Graphene: Fine-grained io management for graph computing. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 285–300. USENIX, 2017.

[33] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[34] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 195–207, 2017.

[35] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 527–543. ACM, 2017.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 135–146. ACM, 2010.

[37] Abdullah A. Mamun and Sanguthevar Rajasekaran. An efficient minimum spanning tree algorithm. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC)*, pages 1047–1052, 2016.

[38] Christian Mayer, Muhammad Adnan Tariq, Chen Li, and Kurt Rothermel. Graph: Heterogeneity-aware graph

computation with adaptive partitioning. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 118–128. IEEE, 2016.

[39] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47, pages 117–128. ACM, 2012.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471. ACM, 2013.

[41] Tesla NVIDIA. P100. *The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU, White paper, NVIDIA*, 2016.

[42] Tesla NVIDIA. V100. *NVIDIA Tesla V100 GPU Architecture Whitepaper, THE WORLD'S MOST ADVANCED DATA CENTER GPU, NVIDIA*, 2017.

[43] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 51, pages 1–19. ACM, 2016.

[44] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14. ACM, 2018.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 410–424. ACM, 2015.

[46] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488. ACM, 2013.

[47] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 28:1–28:12. ACM, 2015.

[48] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD)*, pages 505–516. ACM, 2013.

[49] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 317–332. USENIX, 2016.

[50] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys*, 50(6), 2018.

[51] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 48, pages 135–146. ACM, 2013.

[52] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 425–440. ACM, 2015.

[53] Keval Vora. Lumos: Dependency-driven disk-based graph processing. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, page 429–442, 2019.

[54] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, page 861–878, 2014.

[55] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 507–522. USENIX, 2016.

[56] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 562–573. IEEE, 2014.

[57] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock:

A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 11–21. ACM, 2016.

[58] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, pages 408–421. ACM, 2015.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–28. USENIX, 2012.

[60] F. Benjamin Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of Geographic Information and Decision Analysis*, 1(1):70–82, 1997.

[61] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 207–216. USENIX, 2017.

[62] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 183–193. ACM, 2015.

[63] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 608–621, 2018.

[64] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 441–452. USENIX, 2018.

[65] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (USENIX FAST)*, pages 45–58. USENIX, 2015.

[66] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[67] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316. USENIX, 2016.

[68] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 375–386, 2015.

[69] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 712–725, 2019.

Peregreen – modular database for efficient storage of historical time series in cloud environments

Alexander A. Visheratin
ITMO University

Alexey Struckov
ITMO University

Semen Yufa
ITMO University

Alexey Muratov
ITMO University

Denis Nasonov
ITMO University

Nikolay Butakov
ITMO University

Yury Kuznetsov
Siemens

Michael May
Siemens

Abstract

The rapid development of scientific and industrial areas, which rely on time series data processing, raises the demand for storage that would be able to process tens and hundreds of terabytes of data efficiently. And by efficiency, one should understand not only the speed of data processing operations execution but also the volume of the data stored and operational costs when deploying the storage in a production environment such as cloud.

In this paper, we propose a concept for storing and indexing numeric time series that allows creating compact data representations optimized for cloud storages and perform typical operations – uploading, extracting, sampling, statistical aggregations, and transformations – at high speed. Our modular database that implements the proposed approach – Peregreen – can achieve a throughput of 3 million entries per second for uploading and 48 million entries per second for extraction in Amazon EC2 while having only Amazon S3 as storage backend for all the data.

1 Introduction

Time series data plays a very important role in the modern world. Many fields of science and industry rely on storing and processing large amounts of time series – economics and finances [6], medicine [7], Internet of Things [10], environmental protection [11], hardware monitoring [9] and many others. Growing industry demand for functional capabilities and processing speed led to the sharp rise of specialized time series databases TSDB [16] and development of custom TSDB solutions by large companies, e.g. Gorilla [18] by Facebook and Atlas by Netflix.

One of the most challenging applications of TSDBs, which becomes especially demanded by industrial companies like Siemens, is processing of long periods of historical time series data. Historical data can contain tens and hundreds terabytes of data thus making usage of in-memory databases too expensive. Standard solutions like relational databases or key-value storages struggle to efficiently process such large amounts of time series data [8, 23]. Thus it is very important for a TSDB to achieve the best balance between execution speed and operational cost.

A platform for historical time series analysis groups the data by sensors – modules responsible for capturing one specific phenomenon, e.g. tickers in stock market, temperature sensor for industrial machines or pulse sensor in a hospital equipment. There can be distinguished several typical scenarios of users interaction with the platform for historical data analysis. In the first case, user browses the whole dataset containing millions of sensors and years of data. User can select any sensor to investigate its behavior at any period of time. The platform must visualize the data for the user and allow to quickly zoom in and out, e.g. from years interval into months into days and hours and backwards. To successfully perform in this scenario, the underlying time series storage must provide fast access to any part of the data and be able to extract aggregated values when visualizing long time intervals.

In the second case, user analyzes a specific sensor and wants to look into time intervals, in which the value satisfies some condition, for example amplitude search or outliers search. For this user first searches for all time intervals meeting the condition and after that he or she selects the interval of interest and the platform visualizes the data for this interval. For this scenario the underlying time series storage must be able to perform fast search for time intervals in the whole volume of data and perform fast extraction – loading time series for a specified period – along with aggregation (e.g. average and standard deviation) and transformation (e.g. moving average).

Another important feature for any data processing system is an ability to use it in cloud platforms, like Amazon Web Services, Google Cloud or Microsoft Azure, since they allow the system to be globally available and increase the number of potential users and customers. Considering this the target TSDB must provide a good compression of the raw data along with the small internal footprint to minimize the cost of storing the data in cloud platform. Additional advantage for the target TSDB would be an integration with cheaper data storage cloud services, e.g. Amazon S3 or Google Storage.

Summarizing the above, we faced following challenging requirements for the target time series storage in the beginning of the joint project between Siemens and ITMO University:

1. Store terabytes of time series data for tens of years.
2. 1 second for conditional search in whole database.
3. Execution speed of 450,000 entries per second per node.
4. All data must be stored in Amazon S3.

To address described above requirements we propose an approach for storing numeric time series that is based on twofold split of the data into segments and blocks. Statistical information about segments and blocks is incorporated into three-tier indices, which are independent for every stored sensor. For building an internal data representation we use read-optimized series encoding that is based on delta encoding and Zstandard algorithm and achieves up to 6.5x compression ratio.

We also present design and implementation of the modular distributed database based on the proposed concept. The database is called Peregreen in honor of the fastest bird on Earth. In the experimental studies we demonstrate that Peregreen outperforms leading solutions – InfluxDB and ClickHouse – in the described above use cases. We also show that our solution is able to operate fast in the Amazon EC2 environment while storing all indices and data in Amazon S3 object storage.

Currently Peregreen is being integrated into internal projects of Siemens related to real-time analysis of historical data and development of complex workflows for processing long time series.

2 Proposed approach

The main challenge of development a database that would operate in a cloud environment and store its data solely in a remote object storage is minimizing network overheads. The first aspect of this problem is that there is no reasonable way to perform scans on database files to find appropriate data for the request. Because of that the index of the database has to contain metadata that would allow to navigate to exact parts of database files where the data is located. The second aspect is that we need to minimize the amount of network connections required for data uploading and extraction. For achieving this goal there is a need to develop a format for database files that would allow writing and reading all required data within one request. And the third aspect is minimizing sizes of both database and index files. The smaller files are the cheaper their storage is. And additional advantage of small index files is that more indices can be loaded into the RAM of the processing node.

In this section we give a detailed description of our approach for indexing and storing time series data for fast operation in cloud environments. Target use cases of the initial project included processing of numeric time series that means sequences of pairs *(timestamp,value)* without any tags. To achieve high speed of parallel uploads and ease of operation, every sensor has its own index and processed independently from others. Input time series data is indexed via three-tier data indexing mechanism and converted into the internal representation using read-optimized series encoding. Details on how this approach helps to optimize data processing and fit to the project requirements are given in Section 3.1.

2.1 Three-tier data indexing

The first part of the approach is a three-tier data indexing. Schema of the indexing is presented in Figure 1. At the first step the input data is divided into data chunks – subsets containing *(timestamp,value)* pairs. Time interval for data chunks split is a parameter called block interval. After that for every data chunk we create an index block that contains meta information required for data extraction and a set of metrics – aggregated statistics for the data chunk, e.g. minimum, maximum or average. Metrics are used for conditional search for time intervals in the index, details on it are given in Section 3.1. Block meta includes following fields: number of elements in the data chunk, length of the encoded representation (more details on it in Section 2.2), and a flag whether encoded representation is compressed.

Index blocks are incorporated into an index segment. Blocks in the segment are located in the chronological order, which allows fast calculation of required parts of the segment file as will be described further. During segment creation we initialize all the blocks in it empty, and all changes modify the existing blocks.

The time interval that every segment covers is set by a parameter called segment interval. Every segment has its meta information and a set of metrics. Segment meta contains a unique identifier (ID), start time, which is calculated during segment creation, and a version of the segment. Segment ID sets the name of the segment binary file in the storage backend. The presence of the segment start time makes possible fast and compact encoding of timestamps in blocks as described in Section 2.2.

The start time for a new segment is calculated according to the first timestamp of the indexed data chunk and existing segments in the index. If there are no segments in the index, the start time of the segment is set to the first timestamp of the data chunk. Otherwise, the start time of the segment is calculated so that it is separated by a multiple of the segment intervals from other segments and is as close as possible from the bottom to the first timestamp in the data chunk. The version of the segment is an integer number that increments every time the segment is changed.

Segment metrics are calculated by combining metrics of its blocks. There are two types of metrics – cumulative and non-cumulative. The difference between them is that for combining cumulative metrics there is no need to have all values

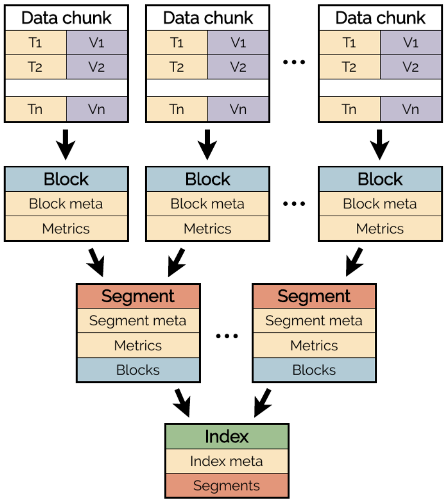


Figure 1: Schema of three-tier data indexing

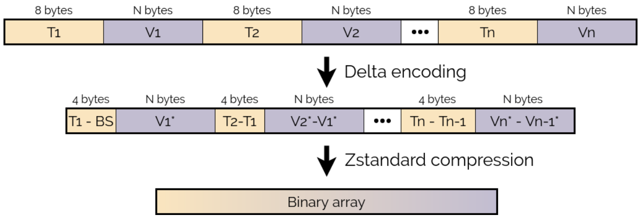


Figure 2: Schema of read-optimized series encoding

based on which these metrics were calculated. For example, minimum and maximum are cumulative metrics, and standard deviation and median are non-cumulative. For cumulative metrics combining function uses only their values while for non-cumulative metrics it uses all input values used to calculate these metrics. Metric instances of blocks and segments store input values during data uploading and delete them afterwards.

Segments are then incorporated into an index. Segments in the index are located in the chronological order. Index meta information includes unique identifier (name of the corresponding sensor), data type, segment interval, block interval and version. Data type determines the way how values are processed during data encoding. Supported data types are 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float and 64-bit float. Block and segment intervals define time intervals for blocks and segments generation as it was discussed above. The version of the index is an integer number that increments every time the index is changed.

Indices are stored in the storage backend as individual files. File names for each individual index consist of index ID and index version.

2.2 Read-optimized series encoding

Since one of the most challenging requirements for our database was storing all data in Amazon S3, we needed an internal representation format that would allow performing data uploading and extraction in the best possible way. It is well known that one of the main factors that affect the speed of interaction with remote services is the network (connections instantiating, latency, etc.). That is why we aimed to create a format that would minimize the number of requests to the storage backend during extraction as much as possible.

The schema of the developed read-optimized encoding is presented in Figure 2. As the input we have a sequence of *(timestamp,value)* pairs from the data chunk. Timestamps are 64-bit integers and thus have the size of 8 bytes. The binary size of values **N** depends on the underlying data type and thus can be from 1 to 8.

The first part of the series processing is the delta encoding – instead of timestamps and values we store differences between current and previous items. It allows to reduce the amount of data required for storing timestamps from 8 to 4 bytes by assuming that difference between two consecutive timestamps would not be greater than ≈ 49 days ($(2^{32}-1)$ ms, maximum value of unsigned 32-bit integer). Delta for the first timestamp is calculated as a difference between the timestamp and block start (**BS** in the schema) that is calculated from the segment start time and the order of the block corresponding to the data chunk. The way how the algorithm generates stored differences for values depends on the processed data type: for integers diff is calculated as a subtraction of the next value from the previous one, and for floating point values a difference is generated as a difference between IEEE 754 binary representations of consecutive values, because such approach produces longer zero-filled bit sequences and can be compressed better.

At the second step calculated deltas are written in the same order as input values into the binary array (see Figure 2). At this point we already get some level of compression. And to get even more we use a high-performance algorithm Zstandard for compression of the obtained binary array. Our experiments shown that usage of the compression adds 10-15% overhead for data reading.

As a result we get a highly compressed representation of the input data chunk – data blocks. During uploading data blocks are written successively into a single array – data segment – that is written as a file into storage backend.

2.3 Discussion

It can be noted that the way of data organization in the presented approach resonates with columnar data formats, like Parquet and Apache ORC, or storage schema of other time series databases [12]. The most distinct and significant difference is that timestamps and values in our case are stored

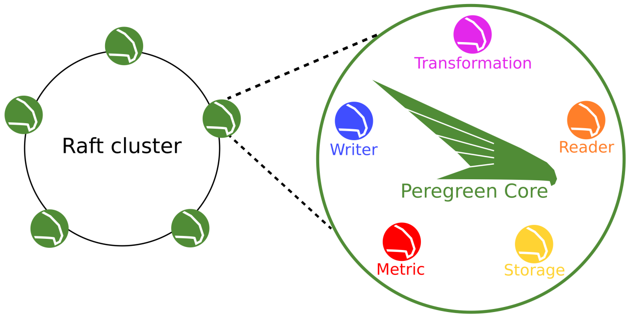


Figure 3: Overview schema of Peregreen. Every node in the Raft cluster includes Peregreen core and five modules.

not separately in different columns/series but together, one *(timestamp,value)* pair after another. This approach along with utilizing binary offsets allows the required parts of segment files to be extracted in a single I/O operation instead of many for other formats (timestamps reading and values reading). When using remote storage backends (Amazon S3 or HDFS) these additional I/O operations significantly decrease performance of data extraction. A possible logical drawback of our approach is a lower level of compression because of mixed sequences; however, experiments described in Section 4.1 show that read-optimized data encoding is more compact than in other solutions.

Another possibly questionable aspect of the proposed approach is the index organization when blocks in a segment and segments in an index are stored in the array rather than access-optimized data structures like trees. During development we experimented with different ways of organizing tiers in the index. We discovered that when having a reasonable number of blocks per segment and segment per index (up to 10,000) array-based index provides almost the same performance as B-tree-based index in terms of elements filtering. This is due to the simplistic logic of the array-based index and absence of recursion calls during the search.

3 Peregreen overview

Peregreen is a distributed modular database that was built to satisfy requirements described in Section 1. Peregreen was developed using Go programming language. Peregreen architecturally consists of three parts – core, modules, and cluster. This section describes these parts and gives information about limitations of the current implementation of the database. Overview schema of Peregreen is presented in Figure 3.

3.1 Peregreen core

Peregreen core is a backbone of the whole system. It implements concepts described in the previous section in a module responsible for the following CRUD operations.

Uploading. Data uploading procedure combines the three-tier indexing mechanism and read-optimized series encoding. At the first step input data is split into data chunks. If there is an existing data for the time interval covered by the new data, Peregreen core extracts it from the storage backend and combines new and existing data. After that the core goes through data chunks and simultaneously converts them into data blocks as described in Section 2.2 and creates index blocks as described in Section 2.1. The core then combines data blocks into segment files and index blocks into index segments. New version of the segment is embedded into the name of the segment file. After combining index segments into the index, segment files and the index are written to the storage backend.

Deletion. On deletion user specifies a time interval, in which the data must be deleted. Indexing engine extracts the data covering target interval from the storage backend, removes all entries that lie in between the start and finish timestamps, updates index and internal representation and loads them into the storage backend.

Search by value. This operation is designed to extract time intervals, values in which satisfy some conditions. Distinctive feature of search operation is that it does not access the data stored in the storage backend, but instead it works only with segments and blocks of the index. It allows to perform very fast search with complex queries over large amounts of loaded data. Time resolution of returned intervals equals to the block interval of the index.

During the search Peregreen core scans through segments of the index and checks whether they fall into the search interval. If a segment belongs to the search interval, indexer iterates over blocks of the segment and checks them for both matching to the search interval and to search query. Metrics available for search queries are all metrics that were computed during the uploading. If new metrics are added after the data was loaded, it is possible to recalculate new metrics for the existing data.

Peregreen provides an easy syntax for describing search queries. Queries consist of conditions combined by logical operators. Conditions have the following form:

```
metric operator value
```

where *metric* is a name of the metric in index blocks, *operator* is a conditional operator, and *value* is a value against which a value of the metric will be compared. Conditional operators used are very close to select operators in MongoDB – *lt* (lower than), *lte* (lower than or equal), *gt* (greater than), *gte* (greater than or equal) and *eq* (equal). Logical operators used for conditions combining include & (conjunction) and | (disjunction). For example, the following command: "min lte 200 & max gte 50 | avg eq 75" describes a query for searching time intervals where one of the following conditions is satisfied: (1) minimum metric is lower or equal to 200 and maximum metric is greater or equal to 50, (2) average metric is equal to 75.

Extraction. Peregreen provides a powerful extraction mechanism that allows performing a fast retrieval of large amounts of data along with data sampling, aggregations and transformations. For extraction, the user specifies sensor identifier, start and finish timestamps, partitioning time interval, and aggregation metrics or transformation functions. Four types of extraction are activated depending on these parameters:

1. Full extraction. This extraction is performed when the user does not specify partitioning time interval and aggregation metrics or transformation functions. In this case, the Peregreen core obtains all points from the storage backend.

2. Sampling. This extraction is performed when the user specifies a partitioning time interval and no aggregation metrics. Data sampling is the extraction of points, timestamps of which differ by user-specified interval, from the storage backend. This operation is useful in cases when the user wants to get a general picture of the data for some time interval. For sampling, the Peregreen core calculates time intervals according to start and finish timestamps, and extracts the first point for every time interval.

3. Aggregation. This extraction is performed when the user specifies a partitioning time interval and some aggregation metrics. This extraction provides the user with a deeper understanding of what is happening in the target time interval. For aggregation, the Peregreen core calculates time intervals according to start and finish timestamps, and generates a set of metrics for every time interval. Metrics available for aggregation include all metrics supported by Peregreen.

4. Transformation. This extraction is performed when the user specifies a partitioning time interval and some transformation functions. This extraction is useful when there is a need for modifying the initial form of the data, e.g. rolling average. For transformation, the Peregreen core applies specified transformation functions to values as they are retrieved.

For extracting values from the storage backend, Peregreen core utilizes meta information stored in all three tiers of the index. Based on start and finish timestamps the core calculates which segments have to be used. For this it uses segments' start times and segment interval from the index. For every segment the core then determines which blocks to extract using segment start time, block interval and blocks ordering. After that the core uses length of the encoded representation of blocks in the segment to calculate the range of bytes need to be extracted from the segment file. The core then extracts only the required part of the segment file, and for every data block of that part performs the actions of the encoding in the reverse order – decompress data with Zstandard, convert raw bytes to deltas and apply delta decoding for them. During decoding stage, Peregreen core applies required aggregations and transformations to the data if specified by user.

It can be seen that the way how internal representation in the Peregreen core is organized allows to achieve the optimal interaction with the storage backend by having only one read operation per segment file. This is the reason why the encoding is called read-optimized.

3.2 Peregreen modules

One of the most powerful features of Peregreen is its extensibility. This section describes five types of modules that allow to integrate various types of storage backends, aggregated statistics, transformations, input and output data types. In terms of implementation, modules are programming interfaces that provide a required set of methods.

Storage. *Storage* is a module, which is responsible for a proper integration of Peregreen with the storage backend. These methods include reading and writing indices, writing segment files and reading binary data for specified index blocks. Currently Peregreen contains three storage implementations out of the box – local file system, HDFS and Amazon S3. Local and HDFS storages allow to work with a data placed within a file system or HDFS respectively. They make use of file offsets for navigating in segment files and extracting only required data parts from them during data reading. S3 storage allows working with a data placed in the Amazon S3 object storage. It stores all data in one bucket specified in configuration file. S3 storage extracts only required data parts from segment files stored in S3 using HTTP bytes range headers.

Reader. *Reader* module is responsible for reading data elements from the input stream. It has only one method for reading a single data element. This is related to the internal mechanics of data reading in Peregreen – it creates data parts, which will be then converted into data blocks, on the fly to minimize memory consumption and total number of operations. That is why there is no need for a method to read all input data at once. The way how *Reader* will extract the data is defined by its implementation and two parameters – data type and format. Data type is a name of one of six supported data types – byte, short, integer, long, float and double. Depending on the data type the way of how *Reader* extracts element might slightly change. Format describes the rules of searching timestamps and values in the input stream. Lets examine how it works on the example of *Reader* implementation for CSV format, which is available in Peregreen by default. Its format definition looks as follows:

```
tsPos-valPos-sep
```

where *tsPos* is position of the timestamp in the string, *valPos* is position of the value in the string, and *sep* is a separator symbol. This simple notation allows to cover a wide range of CSV files layouts. Peregreen also supports JSON and MessagePack as input data formats.

Writer. There are four types of data that Peregreen can return on requests – search results, timestamp-value pairs, aggregations and transformations. *Writer* module describes methods for converting these types into desirable output format. At the moment Peregreen supports three types of output formats – CSV, MessagePack and JSON.

Metric. In order to provide a flexible search on the loaded data, Peregreen allows configuring and extending aggregated statistical characteristics calculated for the data with the help of *Metric* module. It describes three methods, by implementing which one can create new custom metric for the storage, – inserting a value to the metric, getting a value from the metric and combining two metrics into one. The latter method is used for creating metrics for long time intervals from metrics for short intervals. Currently Peregreen has implementations for count, minimum, maximum, average, standard deviation, median, percentile and mode metrics.

Transformation. Transformations allow to change the data as it is extracted. To do that the module provides a single method that converts input value in its new form. Specific implementations, like moving average, might have internal structures for storing intermediate values. Currently Peregreen has implementations for absolute value, difference, summation, subtraction, multiplication and moving average transformations.

3.3 Peregreen cluster

Peregreen cluster provides users horizontal scalability and fault tolerance. Our cluster operates over HashiCorp implementation[1] of the Raft consensus algorithm [17]. This algorithm provides consistency of a cluster. It is especially important for Peregreen because to eliminate the need to handle collisions only one sensor update at a time is allowed. That is why on every update we need to know where required indices are located in order to properly update them.

In Raft cluster write requests go through the leader node so Peregreen also has leader node in the cluster. All upload queries go through the leader node to get redirected to the node that will process uploading. Read requests are redirected directly to the responsible node by any of the cluster nodes. During the first start of the cluster, one node starts as a leader and all other nodes join to that node as followers.

Peregreen cluster state consists of three components:

1. List of nodes. It stores information required for nodes to connect with each other and with clients – node name, internal and external addresses and ports.

2. List of nodes per index. It stores information about mapping of indices onto cluster nodes. Every index is replicated to several nodes in runtime to achieve high availability. If the first node in the list is down, the second one will process the request, etc.

3. Number of indices per node. It stores information about how many indices are stored on every node of the cluster. This is required for the load balancing of the cluster.

Raft log in Peregreen stores information necessary for recreation of the cluster state when the node starts, i.e. information about the events that change the cluster state – addition of a new node, failure of a node, uploading of a new sensor and changing the replication factor.

3.4 Peregreen limitations

Since Peregreen and its underlying concepts are purposefully designed for storing large amounts of historical time series, it has a number of limitations when considering a general task of time series processing.

1. Preferably batch uploads. Although it is possible to load points one by one, it would be highly inefficient because on every such operation the whole data segment file will be downloaded from the storage backend and rebuilt. Even considering that every Peregreen node has a in-memory cache, loading single points brings too much overheads. But the main use case for Peregreen is loading large amounts of time series data at once and the longer input sequence, the better overall performance of Peregreen becomes.

2. One sensor update at a time. When leader receives an update request for a sensor, it locks index for that sensor until the request finishes. This approach was introduced into Peregreen architecture to eliminate the need for collision detection and resolution when more than one node modifies the data for some sensor. We will investigate the best practices for collisions resolution and address this limitation in the future.

3. No multi-sensor operations. All operations in Peregreen require the client to specify the target sensor in the request, and there are no options to specify several sensors. This comes from the fact that the project requirements specifically covered single-sensor operations. At the moment clients make several single-sensor requests for Peregreen and due to the high execution speed they still achieve good performance.

4. No SQL-like syntax. Peregreen has quite minimalistic REST API designed to execute CRUD operations as described in Section 3.1. We understand that supporting SQL would extend the number of use cases but integration of SQL into Peregreen would require significant changes to the architecture of the database.

4 Experimental evaluation

This section describes two series of experiments that were conducted to check different aspects of Peregreen performance in a range of conditions. The first experiment was performed in an on-premise environment on a rack of blade servers, whereas the second experiment was set in the Amazon AWS cloud. Data used in experiments was generated based on a sample of real data with a frequency about 1 Hz. We generated 1 year of data for 1000 sensors that resulted in 31.5 billion records and the total volume of 750 GB.

[1] https://github.com/hashicorp/raft

4.1 On-premise experiment

In order to check the performance of Peregreen compared to existing databases, we performed a detailed experimental evaluation of various data processing operations for Peregreen and two other solutions widely used for time series storage – InfluxDB and ClickHouse. InfluxDB [13] is an open-core time series data store that provides high throughput of ingest, compression and real-time querying. As of November 2018 it is the most popular time series storage [15]. ClickHouse [25] is an open source distributed column-oriented DBMS that provides a rich set of features to its users. Due to the MergeTree table engine and time oriented functions, ClickHouse has proven to be efficient for working with time series data [5,20].

Hardware setup for the first experiment consists of IBM blade servers, each of which has the following characteristics: 2x Intel Xeon 8C E7-2830 (32 vCPUs), 128 GB RAM, 100GB SSD local storage and 2000 GB attached storage from IBM Storwise V3700 storage system connected via fiber channel. All databases were deployed at the local storage whereas the data was stored in the attached storage. In our experiments we used 1, 4 and 7 instances of blade servers.

InfluxDB configuration. In our experiment we used default configuration of InfluxDB provided by its developers with two minor changes. The first is that limit on maximum size of the client request was disabled to upload historical data split by months. And the second, to perform search by values in more equal conditions we created additional aggregation table, which contained 1 hour time intervals and minimum and maximum values for these intervals. Schema of both main and aggregation tables can be found in the listing[2].

When performing several SELECT requests for different time intervals there are two options for InfluxDB – several individual requests or one request with subrequests. Our empirical study[3] shown that making several simultaneous requests is faster than one request with the same number of subrequests. That is why in our experiments we abandoned usage of subrequests.

It must also be mentioned that the trial enterprise version of InfluxDB cluster we used in experiments could not be deployed on 7 instances and because of that experiments for InfluxDB were conducted only for 1 and 4 instances.

ClickHouse configuration. ClickHouse was configured to use the following table schema: date Date, timestamp UInt64, sensorId UInt16, value Float64. Partitioning for the table was set by year and month using toYYYYMM function by date column. The primary key and thus physical order of the records was formed by sensorId and timestamp fields. Table for time series indexing was created as a materialized view of the previous table using 'group by' aggregation by date, sensorId, floor(divide(timestamp, <tdisr>)), where 'tdisr' is a block interval equal to 3600000 ms (1 hour).

Table 1: Data upload time for on-premise experiment, minutes

	1 instance	4 instances	7 instances
Peregreen	891	243	150
ClickHouse	530	193	83
InfluxDB	1847	583	n/a

Table 2: Total stored data volume, GB

Peregreen	ClickHouse	InfluxDB
172	301	272

The materialized view had the following schema: date, sensorId, min(timestamp) AS mnts, max(timestamp) as mxts, min(value) AS mnval, max(value) AS mxval. The primary key of the materialized view was formed by sensorId, mnval and mxval fields. For both tables we used MergeTree engine, which provides the best combination of scan performance, data insertion speed and storing reliability.

These two tables were created on each node due to master-master architecture of ClickHouse. Materialized view tables are updated automatically upon insertion into the main tables. Compression method was changed to Zstandard for all nodes. Index granularity of raw tables was left 8192 as it does not contribute significantly to the increase of data size stored on the disk but allows to position more precisely for 1 hour interval queries. Max thread setting was set to 16 for queries with a single client using SET instruction supported by SQL dialect of the ClickHouse. Other settings were left at their default values. To perform queries to ClickHouse, its native tool called clickhouse-client was used. In experiments with multiple clients, multiple processes were spawned each sending its own SQL query. Native format was used by clickhouse-client to retrieve data.

Peregreen configuration. For all experiments Peregreen was configured as follows: block interval was set to 3,600,000 ms (1 hour), segment interval was set to 31,622,400,000 ms (366 days), compression was enabled. With these settings an average block size was 15 KB, and the segment size was about 130 MB.

Data uploading. In this experiment data uploading requests were sent to storages from all instances of the cluster. It means that for setup with 1 instance it sent all 1000 sensors, with 4 instances each instance sent 250 sensors, and with 7 instances each sent 142-143 sensors. It allows to check scalability of the databases in terms of data uploading.

Results of data uploading time for different sets of nodes are presented in Table 1. It can be seen that ClickHouse provides the highest speed of uploading with the rate of 8 GB per minute on 7 instances. InfluxDB uploading speed was slightly less than two other solutions, but on 4 nodes it demonstrated the uploading rate of ~900,000 records per second, which is quite close to the InfluxData official benchmarks [8].

Table 2 shows the total volume of internal data representa-

[2] https://bit.ly/2P8d4B7
[3] https://bit.ly/2KIKTrG

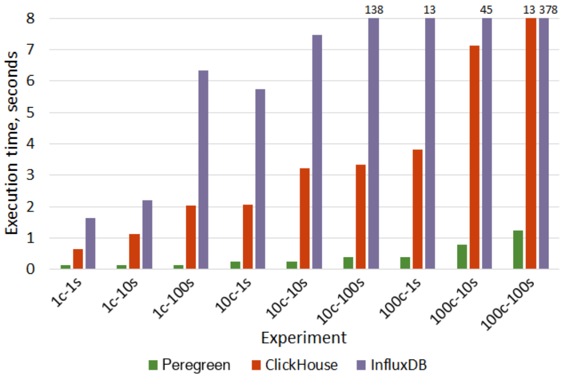


Figure 4: Values search execution time for 4 instances

tion stored by three solutions. With the help of delta encoding and Zstandard compression Peregreen is able to compress the raw data 4 times to 172 GB from 750 GB. Other storages also provide quite high compression rates – 2.5x for ClickHouse and 2.7x for InfluxDB. Despite using Zstandard algorithm, compression rate for ClickHouse is lower than demonstrated in other benchmarks [1]. But compressibility heavily depends on the data characteristics (schema, frequency, etc.) and thus direct comparison of two benchmarks for different datasets would be far from the mark.

It also should be mentioned that indices for all 1000 sensors in Peregreen have the total size of 110 MB, 110 KB per sensor. Such small size ensures minimal footprint of the system and allows to store all indices in the memory in the runtime.

Values range search. In this series of experiments we simulated behavior of clients who try to perform various search by values requests. Number of clients varied in range $[1,10,100]$, number of sensors for which search requests were made by each client varied in range $[1,10,100]$. Every request was amplitude search, where minimum and maximum values cover all values in the dataset, so all solutions had to perform full scan of their storages. In Figure 4 results of this experiment for 4 instances are presented. Columns captions encode experiment types – c stands for the number of clients, s stands for the number of sensors per client. From the figure we can clearly see that Peregreen significantly outperforms two other solutions in all scenarios. This is related to the fact that search operations in Peregreen are performed using only indices, which are stored in the memory, whereas ClickHouse and InfluxDB select results from the tables that are stored on the disk.

Data extraction. These experiments were designed to investigate how well target storages can handle data extraction workloads of various sizes. Number of clients for this series varied in range $[1,10,100]$, number of sensors for which extraction requests were made by each client varied in range $[1,10,100]$, time interval for extraction varied in range $[1,24,240]$ hours. The largest case – 100 clients, 100 sensors per client, 240 hours – is a very extreme scenario that involves extraction of 8.6 billion records.

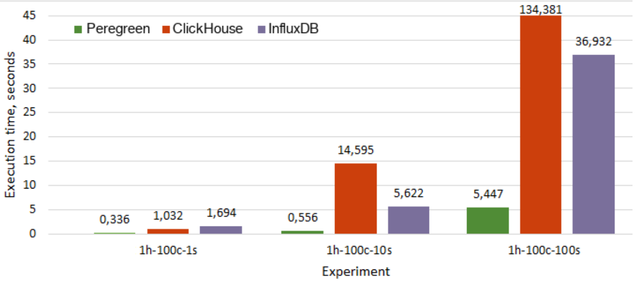


Figure 5: Data extraction time for small requests on 1 instance

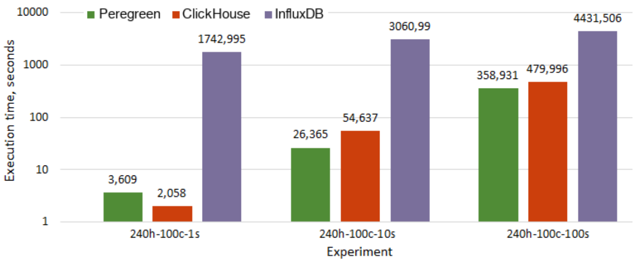


Figure 6: Data extraction time for large requests on 1 instance, logarithmic time scale

Sample results for small requests (1 hour, 100 clients, 1, 10, 100 sensors per client) execution on 1 instance are presented in Figure 5. Columns captions encode experiment types – h stands for the extracted time interval, c stands for the number of clients, s stands for the number of sensors per client. Execution time for Peregreen is much lower than for other systems, because it quickly navigates in segment files using the index and extract only required data blocks (no more than two for these experiments). InfluxDB in these experiments significantly outperformed ClickHouse, which is expected because InfluxDB is designed for working with short time series.

But the situation dramatically changes if we try to extract a lot of long time intervals. In Figure 6 results for large requests (1 hour, 100 clients, 1, 10, 100 sensors per client) are presented. Plots were generated using logarithmic time scale because results differ by orders of magnitude. InfluxDB demonstrates very low extraction speed, since extraction of this many long time intervals is far from its standard use case. ClickHouse, on the other hand, is very good at reading long sequences of data and because of that it demonstrates impressive results of 43,000,000 records per second for the scenario 240h-100c-1s. Although Peregreen performs almost twice slower in this scenario, with increase of parallel requests it starts to outperform ClickHouse and for the hardest case it demonstrates execution speed of 24,000,000 records per second against 18,000,000 for ClickHouse.

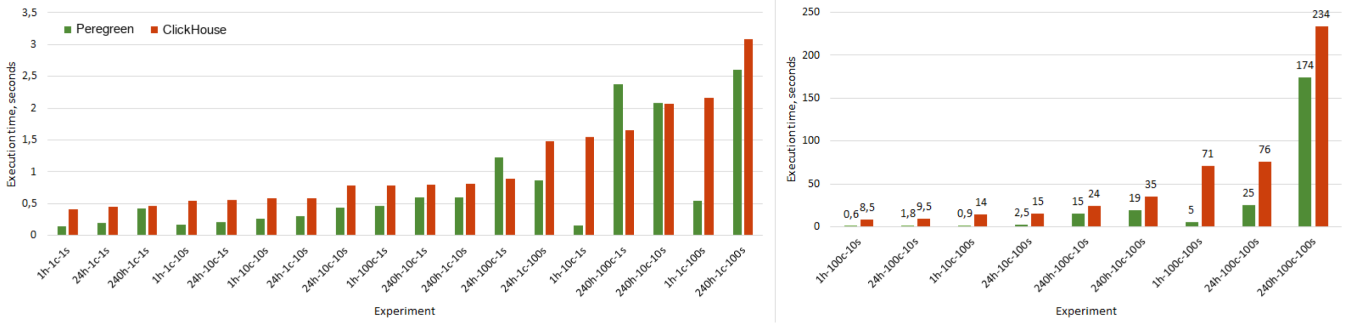


Figure 7: Data extraction time for 7 instances

In Figure 7 experimental results for all types of requests are presented. These experiments were conducted on 7 instances and because of that there are no results for InfluxDB. For relatively small requests, which are shown on the left plot, Peregreen and ClickHouse demonstrate comparable performance. But as the workload grows transcendence of Peregreen also grows. In the hardest scenario 240h-100c-100s Peregreen achieves the speed of 49,500,000 records per second. The speed of ClickHouse is slightly lower, but also very high – 37,000,000 records per second. Difference in the processing speed can be explained by following factors: (1) ClickHouse has lower compression rate (difference in blocks organization and no delta encoding) and because of that has to read more data from the disk during extraction; (2) data parts in ClickHouse are organized by size, not by time, that is why ClickHouse has to read some unnecessary data during extraction by time intervals; (3) ClickHouse creates large number of files with data parts and during extraction it has to read from a lot of files, which increases a number of random reads from disk, whereas Peregreen reads from very small number of segment files.

4.2 EC2 experiment

The second series of experiments was aimed to evaluate performance of Peregreen in a cloud environment for two types of storage backend – local file system and Amazon S3.

Hardware setup for this experiment consists of Amazon EC2 instances of c5.4xlarge type with 16 vCPUs, 32 GB RAM, 8 GB internal SSD storage and 500 GB attached EBS storage. All instances were localed in us-east-1 region. Peregreen was deployed on the local storage whereas the data was stored in the attached storage. Testbed contained 1, 4 and 7 instances.

Benchmarking utility was deployed on a dedicated instance to avoid its influence at performance of Peregreen nodes.

Data uploading. This experiment was designed the same way as in Section 4.1 – data is loaded from all instances of the cluster. Experimental results are presented in Table 3. We can see that in all cases uploading to the cluster with EBS storage backend is faster than to the one with S3 storage backend. It is expected because in the second case segment files are uploaded into remote web service. Nevertheless, both storages provide high uploading rates, 21-23 MB/s per node for EBS storage and 17-18 MB/s per node for S3 storage. Also, both storages demonstrate close to linear change in uploading time with increasing in number of instances, which is very important for the solution deployment on a large scale.

Table 3: Data uploading time for EC2 experiment, minutes

	1 instance	4 instances	7 instances
EBS backend	573	122	75
S3 backend	702	175	97

Search by values. In this series of experiments we performed various amount of parallel search by values requests. Number of parallel requests varied in range $[1, 10, 100, 1000, 10000]$. Since search by values request does not depend on storage backend, we did not compare setups with different backends, and only checked how well Peregreen handles increasing load. In order to make Peregreen process all segments and blocks, we used the following condition in search requests:

```
min lte 500 & max gte -200
```

Due to the fact that searching through indices located in RAM is a very lightweight operation, results for setups with 1, 4 and 7 did not differ significantly. In Figure 8 experimental results for 4 instances are presented. It is clear that search request scales very well and provides a sub-second execution time even for 10000 simultaneous requests.

Data extraction. In this series of experiments we investigated, how well can Peregreen with different storage backends perform under wide range of data extraction requests, and whether usage S3 as a persistent storage of large amounts of time series data can bring performance comparable to the EBS storage. For this we varied number of concurrent requests made to the manager node in range $[1, 10, 100, 1000, 10000]$ and extracted time interval in range $[1, 6, 24, 240]$ hours. Due to the extremely large memory requirements and limited

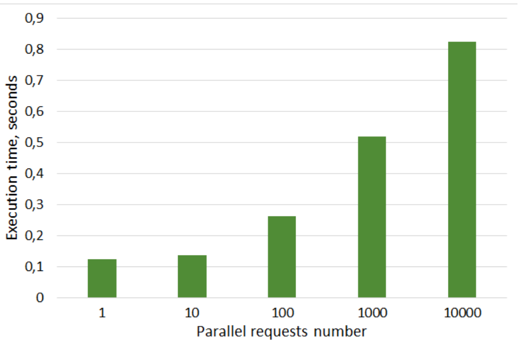


Figure 8: Data search execution time for 4 instances in EC2

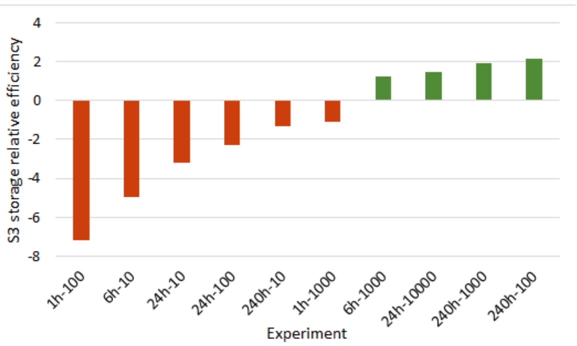


Figure 9: Efficiency of S3 storage backend with regard to EBS storage backend for 4 nodes in EC2

amount of RAM on EC2 instances (320 GB in total) maximal request (10000 requests for 240 hours each) could not be executed, but all other requests completed successfully. All requests were performed 10 times to downplay possible network and hardware effects.

To compare performance of Peregreen with S3 storage backend and with EBSstorage backend we performed all described above experiments in both settings and calculated relative efficiency of S3 as a ratio of execution time with S3 backend to execution time with EBS backend. In Figure 9 S3 relative efficiency plot is presented. Experiment with 100 parallel extraction requests for 1 hour shows the worst S3 efficiency – 49 ms for EBS backend and 351 ms for S3 backend. But as the load (number of parallel requests and interval length) grows, S3 relative efficiency increases, and starting from 1000 requests for 6 hours S3 becomes more efficient than EBS (1320 ms for S3 and 1605 ms for EBS). Maximal loads (100 and 1000 requests for 240 hours) demonstrate two times lower execution time for S3 storage backend – 35096 ms for EBS backend and 18156 ms for S3 backend for 1000 requests. Such difference can be explained by the fact that EBS volumes have limits on a throughput and number of I/O operations per second [2]. S3, on the other hand, does not have such limitations [3], which allows it to scale better under the high workload.

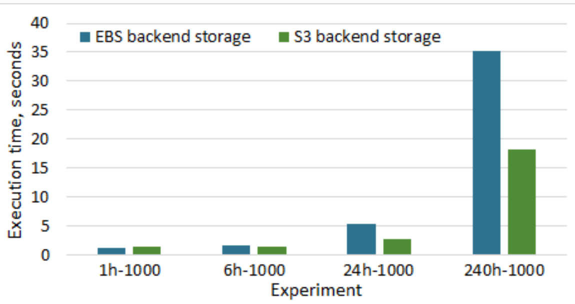


Figure 10: Data extraction results for 4 instances in EC2

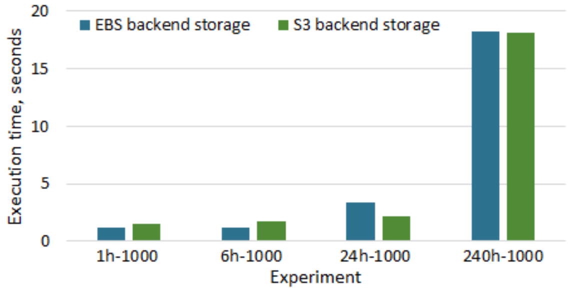


Figure 11: Data extraction results for 7 instances in EC2

In order to further investigate scaling of two storage backends, we analyzed performance of the same set of requests for different sets of nodes. In Figures 10 and 11 we present experimental results for 1000 parallel requests for extraction of 1, 6, 24 and 240 hours on 4 and 7 instances respectively. It is interesting to mention that for 6h-1000 scenario on 4 instances S3 storage backend is slightly more efficient than EBS (1320 ms and 1605 ms), whereas in case of 7 instances EBS backend provides smaller execution time (1233 ms and 1782 ms). But the most important observation is in the 240h-1000 scenario. When Peregreen deployed on 4 instances S3 provides twice faster execution, but for 7 nodes EBS storage demonstrates the same result as S3 storage, and execution time for S3 storage does not change. The reason why this scenario cannot be finished in less than 18 seconds is a network bandwidth limitations – this request generates 864 million data records with the volume of almost 18 GB. Even with a very high EC2 inter-instance throughput [4] it takes a fair amount of time to transfer such data volume to the testing instance.

5 Alternatives comparison

Today there are plenty of time series oriented databases and storages. In this section, we give a brief description of the most popular and mature solutions, which are used for storing time series data, and investigate how they compare against Peregreen. Comparison criteria are based on requirements

Table 4: Comparison of databases for storing time series

	InfluxDB	ClickHouse	Kudu	Cassandra	Gorilla	Gnocchi	TimescaleDB	OpenTSDB	Snowflake	Prometheus	Peregreen
Index type	sparse	controlled sparse	precise	precise	sparse	external	various	sparse	no	sparse	controlled sparse
Internal data structures	LSM+ MVCC	LSM	LSM+ MVCC	LSM+ MVCC	PSP	n/a	B-tree+ MVCC	n/a	tables	custom	custom
Data layout	column	column	column	row-wise	ts-specific	n/a	row-wise	column	column	ts-specific	ts-specific
TS-specific compression	yes	no	no	no	yes	n/a	yes	yes	no	yes	yes
Values-based index	mat. view	mat. view	no	special index	no	no	special index	no	no	special index	special index
Aggregations	yes	yes	no	no	no	yes	yes	yes	yes	yes	yes
SQL-like query syntax	yes	yes	no	yes	no	no	yes	no	yes	no	no
In-memory	no	yes	no	no	yes	no	no	no	no	yes	no
Low-cost object storage	no	no	no	no	no	yes	no	no	partly	no	yes

from Section 1 and common use cases of working with time series. Results of the comparison are presented in Table 4.

InfluxDB is an open-core time series data store that provides high throughput of ingest, compression and real-time querying. As of November 2018 it is the most popular time series storage [15]. ClickHouse is an open source distributed column-oriented DBMS that provides a rich set of features to its users. Due to the MergeTree table engine and time oriented functions, ClickHouse has proven to be efficient for working with time series data [5, 20]. TimescaleDB is an open-source extension of PostgreSQL designed to overcome limitations of PostgreSQL when it comes to working with time series. TimescaleDB introduces hypertable – an entity, which has the same interface for the user as a simple table, but internally is an abstraction consisting of many tables called chunks. This mechanism allows avoiding slow disk operations by storing in memory only the chunk of the latest period. Apache Kudu [14] is a column-oriented data store that enables fast analytics in the Hadoop ecosystem. Apache Cassandra is a general-purpose distributed NoSQL storage, the main aims of which are to provide linear scalability and fault tolerance. Gorilla [18] is a fast in-memory time series database developed by Facebook. Gnocchi is an open-source TSDB that aims to handle large amounts of aggregated data. OpenTSDBis an open-source scalable, distributed time series database written in Java and built on top of HBase. Snowflake is an analytic data warehouse provided as Software-as-a-Service (SaaS). It uses custom SQL database engine with a unique architecture designed for the cloud. Prometheus is an open-source systems monitoring and alerting toolkit that has a powerful time series storage engine.

The first criterion for the comparison is an index type. Precise indices in Kudu and Cassandra make possible fast search of exact data entries, but they generally do not increase the speed of long continuous time series extraction compared to sparse indices in other solutions, and moreover, index sizes in Kudu and Cassandra are very large. A sparse index is much more suitable for the extraction of long time series, and control over the sparsity of the index is an additional advantage. Gnocchi relies on external indexing by PostgreSQL or

MySQL. The interesting fact about Snowflake is that it does not have indices and instead relies on data micro-partitioning and clustering [22] that have proven to be very effective on a large scale.

In terms of internal data structures, log-structured merge-tree (LSM) is the most widespread way to store the data. But InfluxDB, Kudu and Cassandra also use multiversion concurrency control (MVCC) to handle concurrent data modifications. And when using MVCC there can exist several versions of the data, and if these versions were not merged into one during the reading operation, the storage will have to merge versions upon reading, which decreases the overall performance. On the other hand, plain sorted partitions (PSP) used in Gorilla, make possible fast navigation and extraction of large amounts of data. OpenTSDB relies on the backend storage, usually HBase, for storing the actual data. Snowflake tables are the part of the proprietary platform thus no detailed information on them is available. The schema of storing the data in Prometheus includes custom data format for chunks and write ahead logs that are optimized for reading. Peregreen also has a custom format based on read-optimized series encoding.

The next criterion is an internal data layout. Row-wise store data layout of Cassandra and TimescaleDB can be efficient for sequential reads only when there are no other columns except timestamp and value. InfluxDB stores timestamps and values as two separate sequences in its blocks and in that sense its layout is also columnar. OpenTSDB stores the data in columnar storages, like HBase or Bigtable, and because of that provides good speed of data extraction. Prometheus [19] as well as Peregreen has time series specific data layout format. Regarding the time series specific compression, general purpose databases do not have special methods for efficient time series compression, like delta or delta-of-deltas compression. It is worth mentioning that TimescaleDB allows to compress the data by chunks but compressed chunks cannot be updated [24].

The next criteria are related to search by values and data aggregations. The best way to perform a fast search based on values conditions is to have some sort of values-based index. ClickHouse and InfuxDB provide such functionality through materialized views and continuous queries. Cassandra and TimescaleDB can easily create a secondary index over the values column and thus provide very fast search. Snowflake relies on clustering and intelligent query optimization for providing high speed of searching across values conditions. Prometheus allows querying the data based on values of the tags. Index of Peregreen stores aggregated metrics for blocks and segments that makes possible fast search in the large data volumes. All solutions except for Kudu, Cassandra and Gorilla, support data aggregation during extraction.

SQL-like query syntax is a great feature for any database since SQL is familiar to many developers and analysts. Prometheus provides a query language called PromQL that lets users select and aggregate time series data. It should also be noted that Peregreen has no support for SQL syntax and has a set of purposefully built operations available through API.

The last set of features is related to the support of various storage backends. Only ClickHouse, Gorilla and Prometheus can store the data in memory to provide the maximal speed of data access. The most interesting criterion is the ability of the storage to use low-cost object storages, like Amazon S3 or Google Cloud, because it is a very important advantage in terms of cost of usage. Apart from Peregreen only Gnocchi support such functionality. Snowflake supports integration with Amazon S3 but in order to efficiently perform operations over the data it have to be loaded into Snowflake tables [21].

As can be seen from the presented comparison, modern databases provide users with great functionality when speaking about working with time series. Nevertheless, Peregreen provides a unique set of features – high performance along with cloud object storage integration – for working with large volumes of historical time series.

6 Conclusion

In this paper we presented Peregreen – high performance storage for numeric time series. Peregreen is based on a three-tier indexing mechanism and read-optimized series encoding, which allows it to achieve high compression rate, small index size and high execution speed of data processing operations. Experimental results show that Peregreen performs better than InfluxDB and ClickHouse in operations of data search and extraction over large amounts of historical time series data. Peregreen also provides integration with Amazon S3 as a backend data storage out of the box, which means that it can greatly reduce the cost of data storing while increasing the overall efficiency compared to storing the data in EBS, as our experiments demonstrate.

7 Acknowledgements

We would like to thank our shepherd Gustavo Alonso and anonymous reviewers for their valuable feedback. This work is carried out in the Siemens-ITMO Industrial Cognitive Technologies Lab and is funded by Siemens.

References

[1] Altinity. Compression in ClickHouse, 2017.

[2] Amazon. Amazon EBS Volume Types, 2018.

[3] Amazon. Request Rate and Performance Guidelines, 2018.

[4] Amazon. The Floodgates Are Open – Increased Network Bandwidth for EC2 Instances, 2018.

[5] Dmitry Andreev. ClickHouse as Time-Series Storage for Graphite, 2017.

[6] W Brian Arthur. Asset pricing under endogenous expectations in an artificial stock market. In *The economy as an evolving complex system II*, pages 31–60. CRC Press, 2018.

[7] James Lopez Bernal, Steven Cummins, and Antonio Gasparrini. Interrupted time series regression for the evaluation of public health interventions: a tutorial. *International journal of epidemiology*, 46(1):348–355, 2017.

[8] Chris Churilo. InfluxDB Tops Cassandra in Time Series Data and Metrics Benchmark, 2018.

[9] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324. ACM, 2011.

[10] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer applications*, 67:99–117, 2016.

[11] Chris C Funk, Pete J Peterson, Martin F Landsfeld, Diego H Pedreros, James P Verdin, James D Rowland, Bo E Romero, Gregory J Husak, Joel C Michaelsen, Andrew P Verdin, et al. A quasi-global precipitation time series for drought monitoring. *US Geological Survey Data Series*, 832(4), 2014.

[12] InfluxData. In-memory indexing and the Time-Structured Merge Tree, 2018.

[13] InfluxData. InfluxDB home page, 2018.

[14] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: storage for fast analytics on fast data. *Retrieved June from http://getkudu. io/kudu. pdf. Pages,, and*, 2015.

[15] Knowledge Base of Relational and NoSQL Database Management Systems. DB-Engines Ranking of Time Series DBMS, 2018.

[16] Knowledge Base of Relational and NoSQL Database Management Systems. DBMS popularity broken down by database model, 2018.

[17] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[18] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[19] Prometheus. Prometheus storage schema, 2020.

[20] Alexander Rubin. A Look at ClickHouse: A New Open Source Columnar Database, 2017.

[21] Snowflake. Bulk Loading from Amazon S3, 2020.

[22] Snowflake. Micro-partitions Data Clustering, 2020.

[23] Timescale. TimescaleDB vs. PostgreSQL for time-series, 2017.

[24] Timescale. TimescaleDB compression, 2020.

[25] Yandex. ClickHouse home page, 2018.

AC-Key: Adaptive Caching for LSM-based Key-Value Stores

Fenggang Wu Ming-Hong Yang Baoquan Zhang David H.C. Du
University of Minnesota, Twin Cities

Abstract

Read performance of LSM-tree-based Key-Value Stores suffers from serious read amplification caused by the leveled structure used to improve write performance. Caching is one of the main techniques to improve the performance of read operations. Designing an efficient caching algorithm is challenging because the leveled structure obscures the cost and benefit of caching a particular key, and the trade-off between point lookup and range query operations further complicates the cache replacement decisions.

We propose AC-Key, an Adaptive Caching enabled Key-Value Store to address these challenges. AC-Key manages three different caching components, namely key-value cache, key-pointer cache, and block cache, and adjust their sizes according to the workload. AC-Key leverages a novel caching efficiency factor to capture the heterogeneity of the caching costs and benefits of cached entries. We implement AC-Key by modifying RocksDB. The evaluation results show that the performance of AC-Key is higher than that of RocksDB in various workloads and is even better than the best offline fix-sized caching scheme in phase-change workloads.

1 Introduction

The persistent Key-Value Store (KVS) has become an indispensable storage engine in many applications [1–4] for its flexibility and scalability. Existing KVSs, e.g., LevelDB [5], RocksDB [6], Cassandra [7], etc., use a Log-Structured Merge (LSM) tree [8] to improve the performance of write operations. However, their read performance is sacrificed because of the log-structured nature of LSM trees, where finding a key by searching several levels could incur multiple storage I/Os [9–11].

Caching is one of the main techniques to improve read performance since workloads usually demonstrate certain amounts of access locality. Studies show that read operations exhibit "hot spots" in enterprise workloads on LSM-tree-based KVSs (LSM-KVS) for both point lookups [12–14] and range queries [15, 16]. In Facebook, some large-scale production use cases of RocksDB exhibit good locality [17]: fewer than 3% of the keys were accessed during a 24-hour UDB workload; in the ZippyDB workload about 1% of the KV-pairs are accountable for 50% of the total Gets.

There are two major unique challenges in designing an efficient caching scheme for LSM-KVS. First, LSM has a multi-level design where the storage I/O saved by each cached key-value pair (*caching benefit*) could be different. The deeper the level where the KV pair resides, the more storage I/Os can be saved if it is cached. Additionally, the DRAM caching size taken by one key-value pair (*caching cost*) is also different with different key and value sizes. It is challenging for the caching scheme to estimate the cost and benefit and make replacement decisions accordingly. Second, the two types of read operations, namely point lookup and range query, exhibit quite different caching requirements. Point lookup prefers caching an individual key-value pair (KV) for space efficiency [6, 7]. If the value is large, another alternative is to cache a key-pointer pair (KP, where the pointer refers to the location of the value in storage) [7]. In contrast, a range query cannot be served by caching sporadic individual keys, so people resort to caching blocks to support range queries [5, 6]. It is difficult to disentangle the trade-offs among caching KV, KP, and blocks, as each of them has certain types of favorable workloads. Additionally, designing an adaptive caching scheme that can deal with dynamic workloads is more challenging.

Existing caching schemes [5–7, 18, 19] only consider one or two types of entries to cache among KV, KP, and block, and they have a fixed allocated cache budget for one type of entry. Therefore, they cannot leverage all of their merits to cope with various workload scenarios, and cannot adjust the caching space when the workload changes. Besides, to the best of our knowledge, there is no existing work addressing the heterogeneous caching costs and benefits in the unique LSM-KVS scenario.

We comprehensively study the trade-offs among caching KV, KP, and blocks, and propose AC-Key, Adaptive Caching for LSM-based Key-Value Stores, to combine their advantages in handling different workloads. AC-Key uses one designated caching component for each type of the entries (KV, KP, and block). The size of each caching component is dynamically adjusted by the proposed *hierarchical adaptive caching* algorithm that uses *ghost caches* to guide the size adjustment. AC-Key leverages a novel *caching efficiency factor* that quantifies the different caching costs and benefits to aid the boundary adjustment among the caching components as well as the replacement decision within each caching component.

We implement AC-Key based on RocksDB [6]. Our benchmark evaluations show that the read performance of AC-Key is higher than that of the default RocksDB by up to 57.1%. In the phase-change workloads, AC-Key can achieve even better performance than the best offline fix-sized caching scheme. We also evaluate AC-Key using YCSB [15] where AC-Key outperforms the default RocksDB by up to 59.9%.

The rest of the paper is organized as follows. We first provide the background and motivation in §2 and §3, respectively. Then we present the design of AC-Key in §4 and analyze the performance of AC-Key in §5. §6 concludes the paper.

2 Background and Related Work

2.1 LSM-Tree-Based Key-Value Store

Popular implementations of LSM-tree-based Key-value stores (LSM-KVSs), such as LevelDB [5] and RocksDB [6], consist of two parts, a memory component and a storage component. The memory component, or *MemTable*, is typically implemented using in-place sorted data structures such as skip-list or B+ tree. The storage component is implemented as levels of files storing sorted runs of key-value pairs compactly. As shown in Fig. 1, one level is partitioned into multiple *sorted string table* files, or *SSTs*. Each SST has a configurable size limit, typically 2MB~64MB.

When the MemTable is full, it will be formatted into an SST and written to the storage component. This procedure is called the *flush* operation. Each level in the storage component has an exponentially increasing size limit (by default 10 times larger than the previous level). Therefore, larger levels will have more SSTs. L_0 SSTs will be merged with the L_1 SSTs when the specified size limit is reached. After that, L_1 SSTs will then be merged with L_2 SSTs, so on and so forth. This is called the *compaction* operation.

Every level is a single sorted run (except L_0) where the SSTs have disjoint key ranges. The sorted run of key-value pairs of an SST are divided into multiple *data blocks*, and the boundary keys between every two adjacent data blocks are stored in an *index block* with the corresponding data block offset within the SST. Besides, SST also contains a bloom filter block (BF block) to determine the existence of a key in this SST hence to save unnecessary storage I/Os. Block is the basic storage I/O unit in LSM-KVS.

There are two types of read operations in LSM-based key-value stores, namely *point-lookup* (or `Get`) and *range query* (or `Scan`). `Get` is to retrieve the value of a specific key in the following sequence: MemTable, every SST in L_0 from the youngest to the oldest, then L_1 to L_N. If the key is found in the MemTable, it will return with the value without any storage access. Otherwise, the key-value store will search the SSTs in the storage component. It will first check the bloom filter of one SST and skip the SST if the bloom filter indicates that the key does not exist. Otherwise, the index block will be read to locate the corresponding data block. Finally, the data block is retrieved and searched for the key. Therefore, a key-value store needs at most three storage I/Os when searching an SST for a specific key.

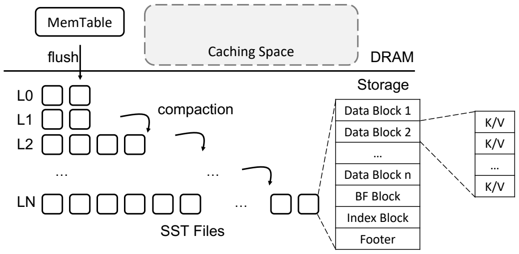


Figure 1: LSM-based Key-value Store (LSM-KVS).

LSM-KVS performs a range query using a starting key and an ending key or a number specifying how many key-value pairs to return. To execute a range query, the KVS uses a `Seek()` function to construct a merging iterator that can iterate through the MemTable, all SSTs in L_0, and one SST in each of the larger levels at the same time. Then, the KVS will call a `Next()` function to return the next larger key. When the key returned by the `Next()` function is larger than the ending key, or the number of the returned key-value pairs has reached the specified number, the range query will be terminated.

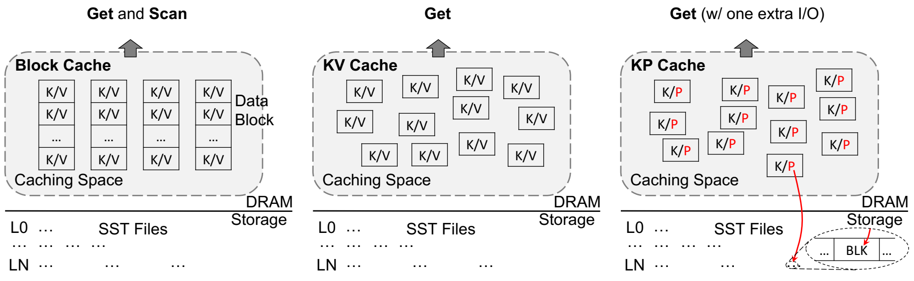
2.2 Related Work

2.2.1 Caching Schemes in LSM-KVS

There are three type of entries that can be cached in LSM-KVS: block, KV, and KP (Fig. 2).

LevelDB [5] only adopts the Block Cache (Fig. 2a), where the blocks could be data block, index block, or Bloom Filter (BF) block. The blocks in the Block Cache are indexed using the SST file ID and the block offset (`<SstID|BlockOffset>`). Block Cache can be beneficial for both point lookup and range query operations. Storage I/Os can be saved as long as the target block is cached. However, Block Cache is not space-efficient to serve point lookup, as the whole block has to be cached even though only a small portion of the keys in the block are accessed frequently.

RocksDB [6] has both Block Cache (Fig. 2a) and KV Cache (Fig. 2b). The KV Cache stores KV pairs that can serve point lookup. However, the sizes of the Block and KV Caches in RocksDB are predefined and fixed. When the KV cache is enabled, point lookup will first consult the KV cache if the write buffer does not contain the key. If the key was not cached in the KV Cache, RocksDB will follow the normal read process using Block Cache as LevelDB does, and insert



(a) Block Cache

(b) KV Cache

(c) KP Cache

Figure 2: Three Types of Entries to Cache in LSM-KVS.

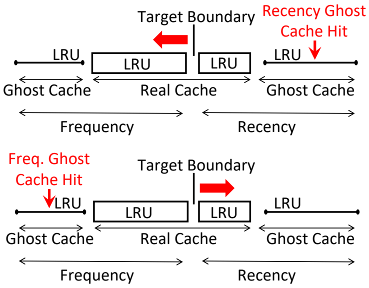


Figure 3: ARC Algorithm.

the KV pair into the KV cache afterwards. Range queries are supported by the Block Cache.

Cassandra [7] does not have a Block Cache but has both KV Cache (Fig. 2b) and KP Cache (Fig. 2c. In the KP Cache, the locations of the values in the storage are cached in memory as pointers). On a hit in the KP Cache, one point lookup can be served with only one storage I/O, skipping all the shallower levels in the storage component. Compared with KV Cache, KP Cache is more space-efficient for large value sizes at a price of one extra storage I/O. Similar to KV Cache, KP Cache cannot handle range queries. When promoting a key from KP Cache to KV Cache, Cassandra does not remove the KP entry.

zExpander [18] is a KVS caching scheme which caches key-value pairs only. It partitions caching space into a compressed zone (Z-zone) and an uncompressed zone (N-zone) and can adapt the boundary between them. LSbM [19] has a small on-disk *compaction buffer* that keeps old, but hot data from being deleted during compaction to reduce the block cache invalidation due to compaction. It needs extra storage space for compaction buffer, and does not take advantage of the more efficient KV or KP Cache to support point lookup.

2.2.2 General Caching Algorithms

The page cache replacement problem has been studied for decades [20]. Adaptive Replacement Cache (ARC) [21] is a dynamic page replacement algorithm designed for managing the page cache in DRAM. As shown in Fig. 3, ARC divides the caching space into two parts, the *recency cache* and the *frequency cache*, each of which is an LRU cache. A page is brought into the recency cache when first encountered. If the page gets a second access before being evicted, it is considered a frequently accessed page and will be migrated to the frequency cache.

The space distribution between the recency cache and the frequency cache is dynamic. ARC uses two *ghost caches* to store metadata of evicted pages (page number) from the recency and frequency cache respectively. The pages stored in the ghost cache will be a future reference for adjusting the allocated space for each part. Compared with the *real cache*, i.e., the cache stores the actual page contents, the size of the ghost cache is negligible since they only store page numbers.

A hit on the recency ghost cache indicates the recency cache should have been larger, so the target boundary will be moved leftwards (top figure in Fig. 3) and vice versa (bottom figure in Fig. 3). As a result, the size of the corresponding real cache will be increased or decreased according to the workload.

CAR [22] also maintains a dynamic partition between the recency and frequency cache using ghost caches, but it uses CLOCK instead of LRU to manage each caching component to reduce overhead. H-ARC [23] uses Non-Volatile Memory (NVM) to cache both clean and dirty pages and propose a hierarchical algorithm to adaptively handle the page replacement.

Such page-based caching algorithms (e.g. ARC [21], CAR [22], H-ARC [23]) do not fit LSM-KVS well because they are based on identical page sizes and caching benefits (i.e. storage I/Os saved by caching an entry), whereas the entry sizes and caching benefits in LSM-KVS are no longer uniform. Similarly, pure frequency-based cache eviction algorithms (e.g. WLFU [24]) and admission policies (e.g. TinyLFU [25]) assume homogeneous entry size and caching benefit too and make replacement decisions solely based on the access frequency. However, in LSM-KVS, the difference in entry size and caching benefit should also be considered along with frequency. Web caching algorithms often take into account the entry size information [26]. In LSM-KVS, however, besides entry size, the caching benefit for different entries is also diverse. For example, caching a KV from a deeper level will save more storage I/Os than caching one from a shallower level. Such special knowledge of the leveled structure of LSM-tree should be exploited to aid caching decisions in LSM-KVS.

3 Motivation

3.1 Unique Challenges in Caching for LSM

Comparing with the page cache replacement problem [20–23], where the pages have identical sizes, keys and values in the LSM tree do not necessarily have the same size. The caching algorithm in LSM should also take the size difference into consideration when designing the replacement algorithm.

Hash-based KVS [27, 28] does not support range queries. In contrast, LSM-KVS has two distinct read operations, point

Table 1: LSM-KVS Caching Scheme Comparison

Cached Entry	Space Efficiency	Extra I/O	Get Existing	Get Missing	Scan	Compaction	Flush	Favorite Workload
Block	Low	No	Helpful	Helpful (BF block)	Helpful	Affected	Not Affected	Scan / Get (missing)
KV	High	No	Helpful	Not Helpful	Not Helpful	Not Affected	Affected	Get (hot/small value)
KP	High	Yes	Helpful	Not Helpful	Not Helpful	Affected	Affected	Get (warm/large value)

lookup and range query, that exhibit quite different caching requirements, and brings additional challenges in the design of LSM caching algorithms.

B+ tree-based KVS [29] supports both point lookup and range query. LSM-KVS is different as it has a leveled structure that diversifies the caching benefit of KV pairs on different levels. The deeper level the KV resides, the more storage I/O can be saved by caching this KV pair. Besides, native operations in LSM-KVS such as compaction and flush do not exist in B+ tree-based KVS, but they will invalidate cached entries and need special treatment in the design.

To motivate our design, we discuss two key question about the LSM-KVS caching: *what to cache*, and *how to perform replacement*, while summarizing the lesson learned.

3.2 What to Cache in LSM-KVS

For point lookup which retrieves the value for a given key, it's natural to directly cache the hot KV pairs in the DRAM cache. However, when the value size is large, and/or the access is less frequent, another alternative is to cache a pointer referring to the location of the value on the storage component. When looking for this key, the value can be fetched using one storage I/O instead of performing multiple I/Os along every SST from L_0 down to the level where the KV resides. By storing a smaller pointer instead of the original larger value, KP Cache can hold more entries. The hit ratio of KP Cache will be higher than that of a KV Cache and can potentially save more I/Os. Comparing KV with KP entry, a hit on a KV entry can save more storage I/Os since a KP hit still needs one storage I/O to get the value. On the other hand, caching KP entries is more space-efficient in the case of relatively large value sizes.

Lesson 1: The merits of caching KV and KP entries should be combined to efficiently serve point lookups.

Unfortunately, cached KV and KP entries cannot help with range queries. Given the staring key of a range query, the next larger key cannot be determined by only examining the sporadic cached KV or KP entries. Therefore, some LSM-KVS implementations such as LevelDB [5] and RocksDB [6] cache data blocks for range queries.

Cached data blocks can serve point lookup too. However, retaining a whole block for point lookup is not space-efficient as it keeps a whole block in the DRAM even only a few keys are frequently looked up. Beside data blocks, frequently accessed index blocks and bloom filter blocks (BF blocks) are also cached in Block Cache.

Lesson 2: Cached blocks and KV/KP entries each have their advantage to support range query and point lookup.

3.3 How to Perform Replacement

From the discussion above, we reach to the conclusion that caching KV, KP, and block each has its own favorable workload scenario. We have a comprehensive comparison summarized in Table 1. However, designing the replacement algorithm for a cache that consists of all these three different entries is challenging.

A straightforward approach is to treat all the cached entries (KP entries, KP entries, or blocks) equally and borrow existing replacing schemes, such as LRU (Least Recently Used) or LFU (Least Frequently Used), to manage the cache. Each entry is inserted or evicted according to the corresponding eviction policy. Therefore, the number of cached entries among the cached KV entries, KP entries, and blocks are solely determined by the access pattern. However, this "unified" caching approach is too simplified and cannot distinguish the differences between these cached entries. First, different cached entries have different sizes. For example, one cached block may take up the DRAM space which can hold tens or hundreds of KV/KP entries. Second, different cached entries have different numbers of saved storage I/Os. For example, one cached block saves one I/O if hit, but one cached KV entry could save multiple storage I/Os for all the SSTs to be accessed for a point lookup. Besides, KV/KP entries on different levels will have different numbers of storage I/O saved too. General cost-aware caching schemes [30, 31] does not have special analysis on the caching cost and benefit in this LSM-KVS scenario either. Third, if there is a single-pass large range query, the fetched blocks will evict useful entries out of cache without bringing any benefit.

Lesson 3: The caching algorithm should consider the difference of DRAM size taken and number of storage I/O saved among different cached entries, respecting the unique leveled structure of LSM-KVS.

Another approach is to have designated fix-sized KV Cache, KP Cache, and Block Cache for the corresponding entries, and perform independent eviction decisions based on popular caching algorithms, such as LRU or LFU. In this case, the cache is resilient to the large single-pass range query (as mentioned before) because the each caching component has a bounded capacity and will not influence the other caching components. However, this fix-sized approach has some problems. First, it is difficult to get the size distribution right in the first place. Second, even if we could have a favorable fix-size configuration at the beginning, it might not be suitable later on as the access pattern of workloads could change over time. For example, a workload has a phase change from range query

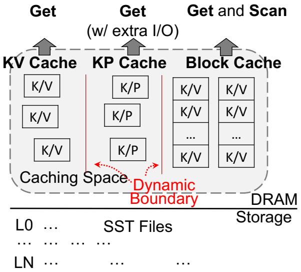


Figure 4: AC-Key Caching Components

dominant into point lookup dominant. If the fixed configuration is good for the range query at first, i.e., majority of the cache space is used as Block Cache, in the point lookup phase, it is clearly not efficient.

Lesson 4: The caching algorithm should be adaptive to the workload changes.

4 AC-Key Design

AC-Key (Fig. 4) caches all three types of entries – KV, KP, and block – with designated caching components for each of the three. Different from the fix-sized scheme described in the previous section, AC-Key has *dynamic* sizes for each of the caching components. The sizes are adjusted by the proposed *hierarchical adaptive caching* algorithm. Considering the heterogeneous costs and benefits of different cached entries and the unique leveled structure of LSM-KVS, AC-Key uses the proposed *caching efficiency factor* to quantitatively guide the size adjustment among the caching components as well as the replacement policy within each caching component.

4.1 AC-Key Caching Components

The AC-Key system architecture is depicted in Fig. 4. The storage component is identical to popular LSM implementations (§2). The DRAM caching space has three components: the *KV Cache*, the *KP Cache*, and the *Block Cache*. The Block, KP, and KV Caches are managed by E-LRU, an improved LRU with cache efficiency factor based eviction (see §4.2).

KV cache stores the key-value pairs directly. KP Cache holds keys with pointers, where the pointers are in a format of `<SstID|BlockOffset>`. When a KP Cache entry is hit, it takes only one storage I/O for the KVS to fetch the data block which contains the target key-value pair. Block Cache stores frequently accessed blocks, which can be either a data block, an index block, or a Bloom Filter (BF) block.

Remarks on the KV and KP Cache. In a point lookup operation, if a *lookup-key* is accessed for the first time, it will be brought to the KP Cache. A key cached in KP cache is called a *warm key*. If a warm key in KP cache is hit again, we consider the key as a *hot key*. We anticipate that it has a higher probability to be accessed again in the future. Therefore, we "promote" the key to KV Cache to potentially save more I/Os from the future accesses. Different from the existing solution in [7] that still keeps key in the KP Cache, AC-Key removes the key from KP Cache to avoid the duplicity and achieve better space-efficiency.

In our current design, we will not "demote" a hot key from KV Cache to KP Cache since we no longer have the pointer information. Another design alternative is to have both the pointer and value stored in KV Cache. However, the accompanying pointer will occupy extra cache space of the KV Cache hence we do not take this approach. As an optimization, if the value size of a KV pair is smaller than the pointer size (implementation-dependent, 24 B in our case), we will cache the *value* with the key into the KP Cache instead of the *pointer* to save the extra I/O without paying more DRAM caching space. This entry still needs another hit to be promoted to the KV Cache. AC-Key does not directly insert this KV entry to the KV cache. The reason is that if this key is not "hot" enough, i.e., having less chance to be hit again, it will evict other "hot" entries from the KV Cache. For example, the workload at a certain time starts to access through a substantial number of keys with small values without a second hit, those keys will occupy the whole KV cache, kicking out useful KV pairs without bringing any benefit.

4.1.1 `Get` Handling

`Get` Existing Key. Denote K as the key to search. First, the MemTable is searched for K as it potentially has the latest version of the value. If not found, then the KV and KP Cache is searched for the key. One of the following cases will happen.

- *Case I*: **Hit in KV Cache**. The value is returned without any I/O incurred.
- *Case II*: **Miss in KV Cache but hit in KP Cache**. Using the cached pointer (`<SstID|BlockOffset>`) as block handle, AC-Key will check whether the data block is already cached in the Block Cache. If not, AC-Key will load the data block into the Block Cache. Using binary search, AC-Key locates the KV pair in the data block and then serves the `Get` request. Besides, the key will be migrated to the KV Cache, promoting the cached entry from key-pointer format to key-value format.
- *Case III*: **Miss both in KV Cache and KP Cache**. AC-Key will examine every sorted run level by level, identifying each SST that has a key range overlaps K, from the youngest to the oldest. Once the key is found in a certain SST, AC-Key will stop searching and return the result directly. Besides, the location of the KV pair, i.e., the pointer, will be recorded and cached in the KP Cache indexed by the key.

In case III, when searching one SST, AC-Key first consults the BF block of this SST. If the BF block is not in the Block Cache, it will be fetched from the storage and inserted into the Block Cache. If the BF block indicates the key is not in the SST, AC-Key will skip this SST and proceed to the next SST. Otherwise, AC-Key will access the index block to narrow down the scope and pinpoint which data block to search. The index block and data block will be fetched from storage and inserted to the Block Cache if not already cached. The BF block, index block, and data block share the Block Cache similar to RocksDB [6].

`Get` Missing Key. When looking up a missing key K in AC-Key, Case I and II of §4.1.1 will not happen as there will be cache miss in both KV and KP Caches.

Similarly, in Case III, the Block Cache will be searched for the corresponding BF block and one storage I/O will be saved if hit. Getting missing keys is the "worst" case where all the overlapping SSTs will be checked. However, by using the cached BF blocks in the Block Cache, multiple storage I/Os can be saved.

4.1.2 `Flush` Handling

`Flush` will dump the MemTable that contains the latest version of the values into the storage component as an L_0 SST. It is possible that a key inserted to the MemTable is already cached in either KV or KP cache. Insertions (also called `Put` operations) to the MemTable will obsolete the corresponding cached entries. As long as the new version of value is still in MemTable, obsolete entries in the caches do not matter because point lookup operation will always consult the MemTable first. However, the cached KV and KP entries must be synced before the keys is flushed to the storage to avoid returning stale results.

We have two alternatives of the timing of the sync: during `Put` or during `Flush`. If during `Put`, the KV and KP Caches will be checked for potential obsolete entries in each `Put` and update the KV entry or delete the KP entry accordingly. Note that during `Put`, AC-Key cannot update the KP entry since the latest value of such key has not been written into an SST yet, hence there is no way to know where the pointer should point to. This approach introduces significant performance overhead because of the extra checking during *every* `Put` operation. Besides, if a key gets multiple updates before flushed from MemTable, we have to repeatedly check both KV and KP Cache for it, which further increases the sync overhead.

Therefore, AC-Key takes another alternative to sync the caches only during `Flush` time. It can accumulate multiple updates to the same key and only sync the KV or KP cache once for each key. Besides, during `Flush` time, AC-Key can figure out the new pointers of keys in the KP Cache and update them accordingly.

4.1.3 `Compaction` Handling

Compaction will affect KP and Block Caches since it creates new SSTs and delete old ones. The old SSTs deleted during compaction may contain blocks that are already cached in the Block Cache. Such deleted SSTs may also contain data blocks being referenced by the pointers cached in the KP Cache. However, it will not affect the entries in KV Cache because compaction reorders and consolidates old KV pairs instead of inserting new ones. AC-Key updates KP and block Caches when compaction affects any of cached KP entries or blocks.

For the KP Cache, during the compaction, AC-Key identifies affected KP entries in the KP-Cache and update the pointers to point to the new data blocks that contain the keys. For the Block Cache, if one cached data block is to be invalidated during compaction, AC-Key will replace the invalidated data block with one new data block generated by compaction to avoid the invalidated block wasting the Block Cache's capacity. The key range of the newly generated data block may not be exactly the same as the old one. AC-Key chooses the block that has the largest overlap with the old cached block and repopulates it back to the Block Cache. Similarly, invalidated cached BF blocks and index blocks are also replaced by the new ones with the most overlapping key ranges. Such block replacement does not incur extra I/O, as the new blocks are generated in memory during compaction.

4.2 Caching Efficiency Factor

To quantitatively analyze the trade-off between the costs and benefits of the cache entries, we propose the novel *caching efficiency factor* that takes into account the unique level structure of LSM-KVS. Using this caching efficiency factor, AC-Key improves LRU into *E-LRU* to manage cache evictions within each caching component, and modifies ARC into *E-ARC* to adjust the size of each caching component. We introduce the caching efficiency factor and E-LRU in this section, and discuss E-ARC in the next section.

We define the *caching efficiency factor E* (E standing for Efficiency) for one cached entry as the following equation. The meaning of this caching efficiency factor is "the number of saved I/O per byte of DRAM caching space".

$$E = \frac{b}{s}, \tag{1}$$

where: E = caching efficiency factor of one cached entry,
b = number of saved storage I/O if cached,
s = caching space taken by this entry.

For example, one typical cached KV entry will take one or several hundreds of bytes, one KP entry will normally take less than one hundred bytes, and one cached block will take 4~16KB. b denotes the number of I/O being saved if this entry is cached. It is given by:

$$b = \begin{cases} 1 & \text{if block,} \\ f(m) & \text{if KV entry,} \\ f(m) - 1 & \text{if KP entry.} \end{cases} \tag{2}$$

where: m = number of SSTs to search for the key,
$f(m)$ = number I/Os to get a key. It is a function of m.

The function $f(m)$ depends on the LSM-KVS implementation. Typically, $f(m) = m+2$, where we have to read m bloom filters each from one SST along the searching path, as well as one index block and one data block in the SST that contains the lookup-key. The number of SSTs to search, m, is estimated by the level l where the key resides:

$$m = \begin{cases} n_0/2 & \text{if } l = 0, \\ l + n_0 & \text{if } l >= 1. \end{cases} \quad (3)$$

where: n_0 = max number of SSTs L_0 can hold,
l = the level where the key resides.

If $l = 0$, the key is in L_0. AC-Key assumes $m = n_0/2$ as an estimate of the average number of SST to search for a key in L_0. If the lookup key resides in levels greater than L_0, potentially every SST in level L_0 will be checked. So, AC-Key uses n_0, the max number of files in L_0, to estimate m.

E-LRU. Traditional LRU only considers the access pattern without taking care of the different benefits and costs of the cached entry. We develop E-LRU, the efficiency-based LRU, to address this issue. E-LRU checks the least used a cached entries and evict one with the least caching efficiency E. The value of a depends on the variance of the caching efficiency factor E of the cached entries. It is given by $a = e^v$, where v is the standard deviation of the caching efficiency factor E of sampled entries in the caching component. When $v = 0$, meaning cached entries has identical efficiency, then $a = 1$, and E-LRU degenerates to the original LRU algorithm that evicts the last one entry from the list. The larger v, the more variance of the efficiency E, the greater a should be used to select a better candidate to evict. In the current implementation, we have a cap on a to avoid AC-Key checking too many entries when making eviction decision. We use E-LRU for simplicity, yet other cost-aware caching schemes [30, 31] can be adapted here using our proposed caching efficiency factor.

4.3 HAC: Hierarchical Adaptive Caching

Hierarchical Adaptive Caching (HAC) has a two-level hierarchy to manage different caching components (Fig. 5). On the upper level, the cache is divided into two components: the Point Cache and the Block Cache. The boundary between the Point Cache and the Block Cache is dynamically adjusted. On the lower level, the Point Cache is further divided into KV Cache and KP Cache with an adjustable boundary too. HAC maintains *ghost caches* to keep a record of the evicted entries from the KV, KP, and Block Cache. On the upper level, there are two ghost caches for the Point Cache and the Block Cache respectively, while in the lower level, there are two ghost caches each for the KV and KP Cache. Opposed to ghost caches, the original KV, KP, and Block Caches are called *real caches*. Here the KV and KP Real Caches collectively make up the Point Real Cache. The ghost caches do not hold the real entry, but only metadata of the evicted entries. A hit in the ghost cache means it could have been a real cache hit if the corresponding real cache was larger. By using ghost cache with the caching efficiency factor, we design E-ARC (caching Efficiency enabled ARC) to adjust the size of the corresponding real cache.

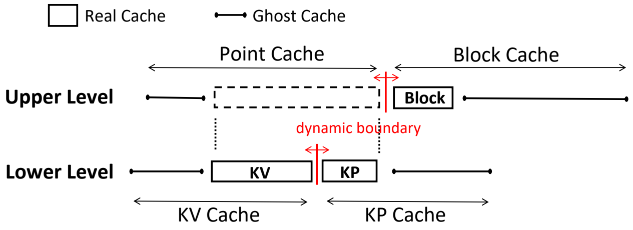


Figure 5: Hierarchical Adaptive Caching (HAC) Algorithm.

4.3.1 Lower-Level HAC

AC-Key uses E-ARC, or efficiency based ARC, to manage the lower-level HAC. On the lower-level HAC, the Point Cache is divided into the KV Real Cache (R_{kv}) and KP Real Cache (R_{kp}). That is: $|R_{kv}| + |R_{kp}| = S_{point}$, where $|\cdot|$ means size, and S_{point} denotes the Point Cache size. AC-Key maintains the KV Ghost Cache as if the KV Real Cache R_{kv} plus the KV Ghost Cache G_{kv} equals to the total size of the Point Cache. Note that the KV Ghost Cache only holds the metadata of the keys evicted from the KV Cache. Denote $|G_{kv}|$ as the size if they were storing the whole KV pair, then $|R_{kv}| + |G_{kv}| = S_{point}$. Similarly, with G_{kp} denoting the KP Ghost Cache, we have $|R_{kp}| + |G_{kp}| = S_{point}$.

Therefore, the following equation is always maintained:

$$S_{point} = |R_{kv}| + |R_{kp}| = |R_{kv}| + |G_{kv}| = |R_{kp}| + |G_{kp}|. \quad (4)$$

We show how E-ARC handles cache hits and misses in the following cases:

- Case I: **Real Cache Hit**. Cache hits on R_{kv} or R_{kp}. Move the hit entry to the MRU end of R_{kv}. Especially, if the hit happens on R_{kp}, AC-Key needs one storage I/O to get the value. Then the key and value is inserted into R_{kv} (promotion).

- Case II: **KV Ghost Cache Hit**. Cache hits on G_{kv} means the size of R_{kv} should have been larger. Shift the target boundary towards the KP Cache end by $\delta = kE$, where E is the caching efficiency factor of the hit entry on G_{kv}. Here k is a configurable learning rate. After fetching from storage, insert the fetched KV to the MRU end of R_{kv}. To make room for this KV entry, evict from R_{kv} (resp. R_{kp}) if the target boundary is within R_{kv} (resp. R_{kp}), meaning that the target size of R_{kv} (resp. R_{kp}) is smaller than its actual size.

- Case III: **KP Ghost Cache Hit**. Cache hits on G_{kp} means the size of R_{kp} should have been larger. Shift the target boundary towards the KV Cache end by $\delta = kE$.

E is the caching efficiency factor of the hit entry on G_{kp}. After fetching from storage, insert the fetched KV to the MRU end of R_{kv}. To make room for this KV entry, evict from R_{kv} (resp. R_{kp}) if the target boundary is within R_{kv} (resp. R_{kp}), similar to Case II.

- Case IV: **Cache Miss**. Retrieve the entry from storage and cache to R_{kp} in KP format. To make room for this KP entry, if the target boundary is within KV cache, evict from R_{kv}. Otherwise evict from R_{kp}.

The target boundary between the KV Real Cache R_{kv} and KP Real Cache R_{kp} indicates the direction the actual boundary should move. The actual boundary will normally lag behind the target boundary. The high level sequence of operations is as follows: 1) ghost hit adjusts the target boundary; 2) entry insertion or promotion shifts the actual boundary towards the target boundary, and as a result, real cache sizes $|R_{kv}|$ and $|R_{kp}|$ are updated; 3) ghost cache sizes are adjusted based on the new real cache sizes using the Eqn. 4; and 4) real and ghost caches perform eviction if necessary to fit the updated sizes using E-LRU (§4.2).

Remarks on E-ARC. Although we follow a similar logic of the canonical ARC algorithm, the original ARC does not have the size and cost differences. In the original ARC, the saved block access b and space cost s for each entry are always the same. E-ARC's definition of $\delta = kE = k\frac{b}{s}$ is a generalization of that of the canonical ARC, and the ARC's definition of the adjustment is a special case of E-ARC's formula where $\frac{b}{s} = 1$.

4.3.2 Upper-Level HAC

On the upper level of HAC, we re-apply the E-ARC scheme to adjust the boundary between Point Cache and Block Cache. Block Cache and Point Cache each has a real cache (R_{block} and R_{point}) and a ghost cache (G_{block} and G_{point}). R_{kv} and R_{kp} collectively forms R_{point}. Blocks evicted from R_{block} enter G_{block}. On the other hand, entries evicted from R_{point} (R_{kv} or R_{kp}) will be inserted to G_{point}. Note that the evicted entry will also be inserted to the corresponding KV or KP Ghost Caches (G_{kv} or G_{kp}) in the lower level (§4.3.1). Similarly to the low level of HAC, the sum of the virtual "size" of the real cache and ghost cache of the Block Cache (resp. Point Cache) will be the total available cache size:

$$\begin{aligned} S_{total} = & \quad |R_{block}| + |R_{point}| \\ = & \quad |R_{block}| + |G_{block}| = \quad |R_{point}| + |G_{point}|. \end{aligned} \tag{5}$$

Target Boundary Adjustment. A ghost hit on G_{block} will move the target boundary between R_{point} and R_{block} toward R_{point} by $\Delta = kE$, where E is the caching efficiency factor of the entry on G_{block} being hit, and k is the learning rate as defined earlier. As a result, the target size of R_{point} will be reduced by Δ. Then, in the lower level, the amount of the adjustment will be distributed between the target size of R_{kv} and R_{kp} proportionally to their current target size ratio. In this example, target size of R_{point} is shrinking by Δ. Denoting the current target size of R_{kv} as $|R^*_{kv}|$ and the target size R_{kp} as $|R^*_{kp}|$, they will be updated as follows: $|R^*_{kv}| \leftarrow |R^*_{kv}| - \Delta\frac{|R^*_{kv}|}{|R^*_{kv}|+|R^*_{kp}|}$, and $|R^*_{kp}| \leftarrow |R^*_{kp}| - \Delta\frac{|R^*_{kp}|}{|R^*_{kv}|+|R^*_{kp}|}$.

On the other hand, a ghost hit on G_{point} will move the target boundary toward R_{block}, i.e., making the target size of R_{block} smaller and that of R_{point} larger. The adjustment amount is also $\Delta = kE$. The caching efficiency factor E of an entry in G_{point} will normally be larger that that in G_{block}, as a Point Cache entry (either KV or KP entry) takes less DRAM caching space and save a greater number of storage I/Os than one Block entry. While there is a ghost hit on G_{point} on the upper level, normally there will be a ghost hit in the lower level too, either in G_{kv} or G_{kp}. In this case, the lower level target boundary will be adjusted first, then that of the upper level. For example, if the ghost hit happens both in the KV Ghost Cache G_{kv} and the Point Ghost Cache G_{point}, the target size of R_{kv} and R_{kp} will first be updated as described in §4.3.1), then increment of Δ is distributed proportionally to the new target size of R_{kv} and R_{kp} accordingly.

Actual Boundary Adjustment. On a block miss when this block needs to be inserted to the Block Cache, if the current Block Cache size plus the new block is greater than the target block size, the Block Cache will not grow. It will evict one block from itself to make space for the new-coming block. Otherwise, if the current Block Cache size plus the new block is within the target Block Cache size, the Block Cache will expand by inserting the new block, and the Point Cache will shrink to make room for the growth of the Block Cache. Typically, more than one KV and KP entries will be evicted because a block is normally larger than cached KV and KP entries. The number of KV or KP entries to be evicted will be based on current target size of the KV and KP Real Caches.

On the other hand, when the Point Cache demands more capacity, (i.e., when new KP entry is inserted to the KP Cache, or when a KP entry is promoted to KV Cache and takes more space), HAC estimates the new Point Cache size after the growth and compares it with the target Point Cache. If the estimated new Point Cache size is within the target Point Cache, one block from the Block Cache will be evicted to make the room for the Point Cache to grow. On the other hand, if the estimated new Point Cache size is above the target Point Cache size, Point Cache will not expand, and the eviction will happen within the Point Cache, following the current target KV and KP Real Cache sizes. One problem is that increasing the size of Point Cache by small amount may result in a whole block evicted from the Block Cache. To address this, not until the target boundary is within the Block Cache for at least one block's size (typically 4KB or 16KB) does HAC evict blocks from the Block Cache. In other words, the size of the Point Cache will grow at the expense of evicting from the Block Cache only when the target Point Cache size is greater by the current actual Point Cache by a whole block size.

4.3.3 Reduce Ghost Cache Size

In the ARC design [21], when a page is evicted from the real cache, the page content is dropped, and only the page number is retained in the ghost cache. The size of the page number is negligible compared to the page content. Similarly, in AC-Key, the ghost cache for the Block Cache G_{block} only stores the block handle in the format of `<SstID|BlockOffset>` (24 B in our implementation) which is also negligible compared to the cache block contents (typically 4~16 KB). However, the ghost caches of G_{point}, G_{kv}, and G_{kp} have significant overhead that can no longer be ignored. For example, assuming a key size of 16 B and the value size of 100 B, one real KV entry takes 116 B. When this entry is evicted from the real cache, the value is dropped, and the key is inserted into the KV ghost cache which still takes 16 B. As the key size is no longer negligible when comparing to the value size and pointer size, a ghost cache potentially occupies a substantial portion of the limited caching space, impair the caching efficiency.

We propose two ways to reduce the space overhead of the ghost caches. First, instead of using the original key of the evicted KV or KP entries, AC-Key only stores a hash value of the evicted key as a "fingerprint". In this way, AC-Key reduces the ghost cache size overhead while sacrificing the ghost cache hit accuracy. Hash collision will cause false-positive ghost hits, resulting in imprecise adjustment decisions. In our implementation, we found a hash value of 4 B shows a good trade-off between the ghost cache overhead and accuracy.

Although using a hash-based fingerprint to replace the key in ghost cache reduces the overhead, a hash value (for example, 4 B) is still taking significant space compared with the real KV and KP Caches. We further propose another optimization to eliminate such ghost cache overhead by disabling ghost cache when the adaptive scheme settles to a favorable capacity distribution among the KV, KP, and Block Cache. If the changes of hit ratios of all the caching components remains within a threshold θ (called *ghost cache turnoff threshold*, 5% by default), the ghost cache mechanism will be turned off, and the space taken by the ghost caches will be reclaimed for real caches. The size of the real caches will be proportionally scaled up to use all the available cache capacity when the ghost cache's space is released. Later, when the access pattern switches and the current size distribution is not favorable, the change in the hit ratio will flag the phase transition. When the hit ratio fluctuates beyond the threshold of θ, the AC-Key will turn the ghost cache mechanism back on until the hit ratio converges again (fluctuating within the threshold θ).

5 Evaluation

5.1 Implementation and Setup

We implement AC-Key based on RocksDB `version 6.2` with roughly 5.6K lines of change in `C++` code. We carry out our experiment on a Dell PowerEdge R430 1U Server. It has two six-core Intel Xeon E5-2620 v3 @ 2.40 GHz processors and 64 GB of DDR3 memory. The operating system is Ubuntu LTS 18.04 with Linux kernel version 4.15.0. The storage device is a 372 GB Intel DC P3700 PCIe SSD formatted as `xfs`.

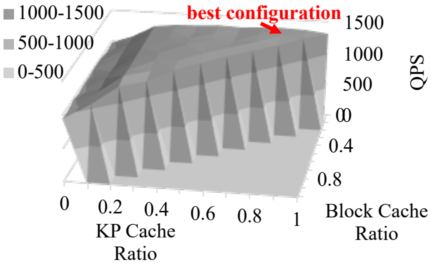


Figure 6: The `offline` scheme tries out on different configurations (at a smallest granularity of 1/10 of the cache size) and select the fix-sized configuration with the best result.

We load a 100GB database with randomly generated keys [9, 11]. The default key size is 16 B and value size is 100 B, which is the default value of RocksDB [32]. The length of range scans is set as 100, which is close to the length used in literature [15, 33–35]. The default point lookup to range query ratio is set to 1:1. We use the existing exponential function based workload generator in RocksDB [32] to generate workloads with different skewness of hot keys (point lookup key and scan starting key). In our experiments, we take the default skewness as the hottest 1% keys taking up the 99% of the accesses (including both the key for point lookup and the starting key for range query). The default learning rate k is 100K (see §4.3) and default ghost cache turnoff threshold θ is 5%. Data compression is disabled to rule out unrelated performance interference and simplify the analysis as in literature [9, 36]. We specify identical cache sizes in the configuration files of RocksDB and YCSB to ensure the same amount of caching budget is used in competing schemes. Page-based direct I/O [37] is enabled to rule out the interference of the OS buffer cache, similar to [38, 39].

The following schemes are compared.

- `pure-kv`: The whole caching space is used as KV Cache.
- `pure-kp`: The whole caching space is used as KP Cache.
- `rocksdb`: Off-the-shelf RocksDB with default setting. Note that RocksDB disables KV cache by default and the whole caching space is used as Block Cache.
- `offline`: We try combinations of different component size with the granularity of 1/10 of the cache size and select the best fix-sized configuration. Note that such best configuration is determined offline, and is not applicable in a real-time caching system (Fig. 6).
- `ac-key`: Our AC-Key scheme.

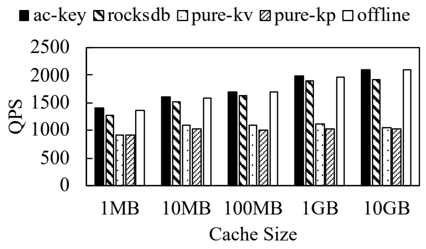


Figure 7: Varying Cache Size.

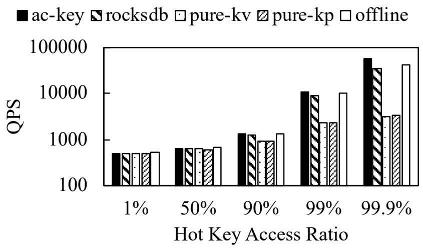


Figure 8: Varying Skewness.

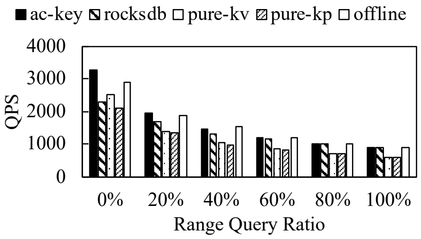


Figure 9: Varying Range Query Ratio.

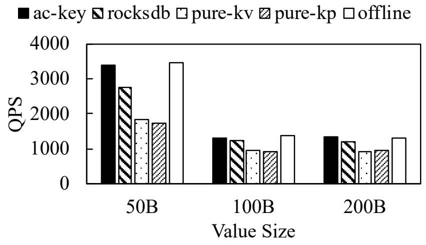


Figure 10: Varying Value Size.

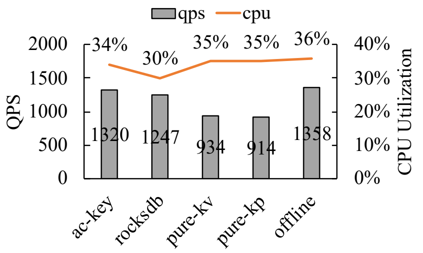


Figure 11: CPU Utilization Comparison.

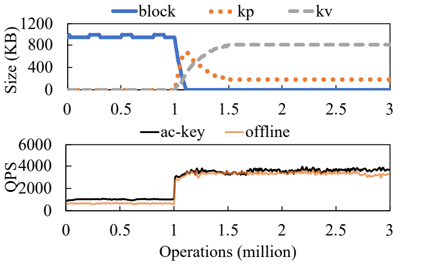


Figure 12: Adaptive Adjustment in AC-Key.

5.2 Micro-benchmark

Varying cache size. We vary the cache size from 1 MB to 10 GB, collect the Query Per Second (QPS) and plot them in Fig. 7. As the cache size increases, the QPS also increases for `ac-key`, `rocksdb`, and `offline`, as larger cache can potentially cache more entries, hence results in less cache misses. In contrast, `pure-kv` and `pure-kp` do not improve much because they do not support range queries, and therefore the range queries will cause many storage I/Os. `ac-key` outperforms other caching schemes that have only a single type of caching component (better than `rocksdb` by 5.0%~9.1%, `pure-kv` by 47.1%~97.7%, and `pure-kp` by 52.8%~104.5%) since they cannot serve both point lookups and range queries efficiently. Comparing with `offline`, `ac-key` performs close to or even better than `offline` because of the ability to adaptively configure the size of each caching component. The reason for the better performance of `ac-key` in some cases is because `offline` cannot exhaust all the possibilities of configurations as `offline` only tries out the cache size at a granularity of one-tenth of the total cache size (Fig. 6).

Varying skewness. We vary the access skewness of the point lookup and the range query using RocksDB's native benchmark tool [32]. In Fig. 8, x-axis shows the access ratio of the hottest 1% ranges from 1% (no skew, uniform distribution) to 99.9% (very skewed). Note that the y-axis is set to log-scale to enhance readability. We can see that when the access is uniformly distributed (left-most cluster of bars) over all the keys, the QPS is similarly low for all the caching schemes. This is because in a uniform-distributed workload, every caching scheme has hardly any hit. In contrast, as the access becomes increasingly skewed, the performance of all the schemes rises, and `ac-key` outperforms `rocksdb` by 3.6%~57.1%, `pure-kv` by 5%~17.6×, and `pure-kp` by 7%~16.5×.

Varying range query ratio. We change the ratio of the point lookup and range query in the workload, ranging the range query ratio from 0% (pure point lookup) to 100% (pure range query) to create Fig. 9. In this figure, we see that as the range query ratio increases, the QPS decreases. This is because a range query has more potential storage I/Os and occupies more caching space. `ac-key` has similar result as `offline` and is better than `rocksdb` (by 1.1%~42.6%), `pure-kv` (by 30.4%~54.7%), and `pure-kp` (by 43.0%~52.8%).

When increasing the range query ratio from 0% to 20%, the performance of `rocksdb` becomes better and exceeds `pure-kv`. This is because KV Cache cannot serve range queries. In contrast, Block Cache supports both point lookups and range queries. However, the space efficiency of Block Cache is low which misses the opportunity to cache more useful entries. `ac-key` adaptively combines Block Cache, KV Cache, and KP Cache and delivers the best performance.

Varying value size. We try different value sizes ranging from 50B to 200B (Fig. 10). As the value size increases, the performance of all caching schemes decreases. This is because larger value size incurs more storage I/O overhead per each key being read. `pure-kv` performs better than `pure-kp` at small value but is exceeded by `pure-kp` as the value size increases. This is because the total number of keys can be cached for `pure-kv` decreases as the value size increases, hence the performance of `pure-kv` becomes not as good as that of `pure-kp`. `rocksdb` still performs better than `pure-kv` and `pure-kp`, as half of the requests are range queries that cannot be served by the KV or KP Caches. `ac-key` performs close to `offline`, and constantly better than `rocksdb` (by 5.9%~22.4%), `pure-kv` (by 41.4%~83.6%), and `pure-kp` (by 44.5%~96.0%) because of the use of the hierarchical adaptive caching in adjusting the size of each component.

CPU Utilization Comparison. With the default settings (§5.1), we record the CPU utilization by the Linux native `time` command and plot them in Fig. 11. The QPS is also plotted to better illustrate the trade-off. We can see that `rocksdb` consumes less CPU resource than the other schemes (`ac-key`, `pure-kv`, `pure-kp`, and `offline`) because they all use Point Cache and thus need to calculate the hash value of *every* key inserted to check if they were cached. We can also observe that although `ac-key` has extra operations to adapt the size of caching components, the CPU utilization is not increased significantly comparing with `pure-kv`, `pure-kp`, and `offline`. Besides, `ac-key` performs close to `offline` and is better than `rocksdb`, `pure-kv`, and `pure-kp`.

5.3 Adaptive Adjustment in AC-Key

To verify the adaptive adjustment process of AC-Key, we construct a workload with two phases: 1 million range queries with random starting key, then 2 million random point lookups. We plot the size of the caching components – Block, KV, and KP Caches – to show the direct evidence of the adaptation process (Fig. 12 top figure). Besides, we also compare the real-time QPS of `ac-key` and `offline` (Fig. 12 bottom figure).

We can see from the top figure of Fig. 12 that during the first 1 million queries, the sizes of the KV and KP Cache are reduced to zero to maximize the Block Cache since they cannot serve range queries. When the workload changes from pure range query to pure point lookup after the first 1 million queries, the Block Cache shrinks sharply, and the KV and KP Cache start to grow. This is because Point Cache (KV and KP Cache) are more space-efficient in caching point lookups, so the HAC algorithm adjusts in favor of the Point Cache (including both the KV and KP Caches). The KP Cache grows faster than the KV Cache at the beginning because every entry is first cached in the KP Cache, and only a small amount of "hot" KVs with a second access will be migrated to the KV Cache. At the beginning of the second phase, both KV and KP Cache grow as they are "stealing" space from the Block Cache. After the size of the Block Cache declines to almost zero, the KV and KP Cache start to compete with each other for space, and it is when KP Cache starts to shrink. The size of the KP and KV Cache finally converges to 19% and 80% of the total cache size. In different workload settings, the Block, KP, and KV Cache will stabilize into different ratio.

`ac-key` is adaptive and can adjust the size of the caching components based on the workload. We also run the `offline` scheme, which tries to find the best fix-sized configuration. However, such one-size-fit-all configuration is not tailored for the special workload of each phase, thus has inferior performance than `ac-key` in each phase: `ac-key` is 32%~66% better in the range query phase, and 2%~20% better in the point lookup phase after convergence (bottom figure in Fig. 12).

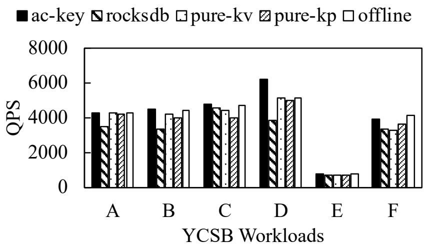


Figure 13: Macro-benchmark YCSB Evaluation.

Table 2: Average Write Latency (μs) and Regression.

Workload	A	B	C	D	E	F
RocksDB	242.8	238.0	N/A	21.7	32.8	28.6
AC-Key	241.1	241.4	N/A	22.7	33.5	26.7
Regression	-0.7%	1.4%	N/A	$\boxed{4.9\%}$	2.0%	-6.8%

5.4 Macro-benchmark YCSB Evaluation

We also use six workloads in YCSB [15] to further verify the performance of our design in near-production workloads. As can be seen in Fig. 13, the overall QPS of `ac-key` is higher than `rocksdb` (by 3.6%~59.9%), `pure-kv` (by 0.1%~20.7%), and `pure-kp` (by 1.2%~25.2%).

We also measure the regression of write performance of AC-Key from RocksDB (Table 2) and find the regression is below 5%. This shows that `ac-key` does not incur significant overhead on the write operations while improving the overall performance.

5.5 Sensitivity on Parameters

Using the default setting in §5.1, we range the learning rate k in the adaptive algorithm (see §4.3) from 1K to 10M, i.e. (1K, 10K, 100K, 1M, 10M). The fluctuation of the QPS is within 3.9%, ranging from 1461 to 1520 operation/s. We also test the ghost cache turnoff threshold θ (§4.3.3) from 0%, 5% , $\cdots$, 30%, and the QPS stays nearly constant (1418~1459, varying within 2.8%). As the two parameters do not have a significant impact on the result, we set their default values as $k = 100K$ and $\theta = 5\%$.

6 Conclusion

Caching is one of the essential techniques to improve the read performance of LSM-tree-based key-value stores. We investigate three different types of entries being cached, namely block, KV, and KP, and incorporate them into one integrated cache. We propose a Hierarchical Adaptive Caching (HAC) scheme to dynamically adjust the size of the block, KV, and KP caching components. To deal with the heterogeneous costs and benefits of the cached entries, we leverage a novel caching efficiency factor to aid the size adjustment among caching components and the eviction decisions within each caching component. We implement the proposed AC-Key by modifying RocksDB. Evaluations show that AC-Key improves the read performance of default RocksDB by up to 57.1% without significant impact on write performance.

Acknowledgments

We thank the anonymous ATC reviewers and our anonymous shepherds for their feedback. This work was partially supported by NSF I/UCRC Center Research in Intelligent Storage and the following NSF awards 1439622, 1525617, and 1812537.

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[4] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[5] Google. Leveldb. https://leveldb.org/.

[6] Facebook. Rocksdb. https://rocksdb.org/.

[7] Apache. Cassandra. http://cassandra.apache.org/.

[8] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[9] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, 2016. USENIX Association.

[10] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, 2016.

[11] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[13] John Liang, James Luo, Mark Drayton, Rajesh Nishtala, Richard Liu, Nick Hammer, Jason Taylor, and Bill Jia. Storage and performance optimization of long tail key access in a social network. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, pages 1–6. ACM, 2013.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[16] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[18] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 14. ACM, 2016.

[19] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. *ACM Transactions on Storage (TOS)*, 14(2):15, 2018.

[20] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[21] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[22] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[23] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[24] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.

[25] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[26] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. page 293–305, 1996.

[27] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *NSDI*, volume 10, pages 29–29, 2010.

[28] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[29] MySQL 8.0 Reference Manual. The innodb storage engine. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[30] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[31] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 327–337. IEEE, 2003.

[32] Facebook. Rocksdb – benchmarking tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[33] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[34] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct stries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336. ACM, 2018.

[35] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[36] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[37] Jonathan Corbet. Page-based direct i/o. https://lwn.net/Articles/348719/, 2009.

[38] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

[39] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520. ACM, 2018.

POSH: A Data-Aware Shell

Deepti Raghavan Sadjad Fouladi Philip Levis Matei Zaharia
Stanford University

Abstract

We present POSH, a framework that accelerates shell applications with I/O-heavy components, such as data analytics with command-line utilities. Remote storage such as networked filesystems can severely limit the performance of these applications: data makes a round trip over the network for relatively little computation at the client. Reducing the data movement by moving the code to the data can improve performance.

POSH automatically optimizes *unmodified* I/O-intensive shell applications running over remote storage by offloading the I/O-intensive portions to proxy servers closer to the data. A proxy can run directly on a storage server, or on a machine closer to the storage layer than the client. POSH intercepts shell pipelines and uses metadata called *annotations* to decide where to run each command within the pipeline. We address three principal challenges that arise: an annotation language that allows POSH to understand which files a command will access, a scheduling algorithm that places commands to minimize data movement, and a system runtime to execute a distributed schedule but retain local semantics.

We benchmark POSH on real shell pipelines such as image processing, network security analysis, log analysis, distributed system debugging, and `git`. We find that POSH provides speedups ranging from 1.6× to 15× compared to NFS, without requiring any modifications to the applications.

1 Introduction

The UNIX shell is a linchpin in computing systems and workflows. Developers use many tools at the command-line level for data processing [33], from core `bash` utilities, including `sort`, `head`, `cat` and `grep` to more complicated programs such as `git` [22], ImageMagick [30] and FFmpeg [4]. Network security engineers use shell pipelines to find potential patterns in gigabytes of logs. The shell's continued importance over many decades and generations of computing systems shows just how flexible and powerful a tool it is.

The UNIX shell, however, was designed in a time dominated by local and then LAN storage, when file access was limited by disk access times, such that the overhead of network storage was an acceptable trade-off. Today, however, solid-state disks have reduced access times by orders of magnitude. At the same time, networked attached storage, especially for the enterprise, remains extremely popular [9, 57, 60]. Mounting filesystems across the wide area could incur tens of milliseconds of latency. Furthermore, many applications use wide area storage systems, via cloud blob storage [25, 54] or cloud-backed filesystems [7, 48, 50, 51, 58].

Running I/O-intensive shell pipelines over the network requires transferring huge amounts of data for little computation. For example, consider generating a `tar` archive on NFS. The `tar` utility effectively copies the source files and adds a small amount of metadata: the server reads blocks and sends them over a network to a client, who shifts their offsets slightly and sends them back. NFS mitigates this problem by offering compound operations [29] and server-side support for primitive commands such as `cp` [41]. However, something as simple as `tar` requires large network transfers.

The result of these changing performance trends is that network transfer is an increasingly large overhead on shell scripts. For example, unzipping a dataset of size 0.5 GB with `tar -x` over NFS within a cloud datacenter takes 7× longer than on a local disk. While intra-datacenter networking is fast, it is not as fast as a local flash drive. Some workflows are so slow over the network that they are effectively unusable: running `git status` on the Chromium repository [24] takes 2 seconds locally, but if the repository is stored in a nearby datacenter, it takes over 20 minutes.

The underlying performance problem of using the shell with remote data is locality: because the shell executes locally, it must move large amounts of data to and from remote servers. Data movement is usually the most expensive (time and energy) part of a computation and shell workloads are no exception. Near-data processing [1, 5, 14, 47, 52, 59] can reduce data movement overheads. Data-parallel processing systems such as Spark [61], stored procedures in SQL databases [43, 44], and native data structures in key-value stores such as Redis [40] all bring computation closer to the data. However, many of these systems require applications to use their APIs: they can supplement, but not replace shell pipelines.

To address the shell performance problem of data locality, this paper proposes POSH, the "Process Offload Shell", a system that offloads portions of *unmodified* shell commands to *proxy servers* closer to the data. A proxy server can run on the actual remote fileserver storing the data, or on a different node that is much closer to the data (e.g., within the same datacenter) than the client. POSH improves shell-based I/O workloads through three techniques. First, it identifies parts of complex pipelines that can be safely offloaded to a proxy server. Second, it selects which candidates run on a proxy, in order to minimize network data movement. Finally, it executes the pipeline across an underlying runtime, stitching together the distributed computations while maintaining the exact output semantics expected by a local program. Correctly and efficiently implementing these three techniques has three principal challenges:

1. Correctly understanding the semantics of shell command-line invocations in order to deduce which files each command in the pipeline accesses and determine which

commands can be offloaded.

2. Distributing the entire pipeline across different machines to minimize overall data movement, based on the "closest" execution environment (client or proxy) for each command and its file dependencies.
3. Automatically parallelizing pipelines that access many files while ensuring the output maintains a sequential execution order.

In order to address the challenge of understanding the semantics of shell command-line invocations, POSH uses *annotations*. POSH's key insight is that many shell applications only read and write to files specified in their command-line invocation, so POSH can deduce which files a command accesses from a model of the application's argument structure. Annotations store a model of each command's (e.g., `cat`'s or `grep`'s) semantics and arguments, stored locally at the client's shell. These annotations, inspired by recent proposals to annotate library function calls for automatic pipelining and parallelization [46], assign *types* to the possible arguments of command-line applications. At runtime, POSH can parse which arguments are files and use the underlying storage configuration to determine where those files are located.

Next, POSH must schedule the entire pipeline across the execution engine in a way that reduces data movement as much as possible. However, POSH does not know explicitly how much data is transferred across each pipe in the entire pipeline. If the output of `grep` or `awk` is piped to another command, POSH cannot know how much data will travel over the pipe without running the command. POSH constructs a DAG representation of the entire command and associated metadata and applies a greedy algorithm, that estimates how much data travels across each pipe, to determine the best way to schedule the DAG.

Finally, POSH further improves performance when it knows it can split commands into multiple data-parallel processes. POSH ensures that each split parallel invocation retains the same argument structure as the original command and that the output is stitched together in the correct order. It does so by using information about each argument stored in the annotation to safely split the command. POSH's execution engine serializes output from any parallel processes in the correct order, before writing to the final destination (e.g., client `stdout`).

This paper makes the following contributions:

1. **An expressive annotation language for shell commands**: Annotations capture the dominant grammar of most shell commands and summarize which inputs to command invocations are files as well as semantics to safely split command invocations across inputs.
2. **A greedy scheduling algorithm that reduces data movement for unmodified shell pipelines**: POSH's scheduling mechanism decides which parts of pipelines can be offloaded to proxy servers closer to the data and which parts of pipelines can be parallelized, while preserving the correctness of the output.

We evaluate POSH on a variety of workloads, including an image processing pipeline that creates thumbnails, a `git` command workflow on the Chromium repository, and log processing pipelines from research project analysis. On one hand, for a more compute-heavy log analysis application that includes a data transfer of one large file, POSH provides a 1.6× speedup over running `bash` over NFS. In the best case, for a `git` command workflow, POSH provides a 10-15× speedup over running `bash` over NFS, even when the client and the server are in the same datacenter. Section 8 contains the set of full results.

2 RELATED WORK

Near-data computing (NDP). POSH draws upon previous work that "ships computation closer to the data." Previous approaches to NDP, surveyed in [5], focus on two paradigms: PIM (processing in-memory), where compute is co-located with memory [28, 45, 47], or ISC (in-storage computing), where processors or accelerators are co-located with persistent memory or storage [1, 14, 52, 55, 59]. POSH follows the second approach and pushes portions of arbitrary shell applications to *proxy servers*, compute units running on general-purpose servers, either co-located with storage devices or closer to storage devices than the client.

Barbalace et al. [5] propose an entire operating system architecture for NDP with features such as locality-driven scheduling. Various systems focus on offloading database query computation into smart SSDs [14] or FPGA accelerators [32, 52, 59]. The Metal FS framework [52] allows users to run reusable compute kernels, that perform operations such as encryption or filtering, on FPGA accelerators near the storage. They can be programmed with standard shell syntax such as pipes to chain many near-data operations together. Seshadri et al. [55] propose an extensible programmable SSD interface where users can push specific pieces of functionality, e.g., to run filesystem appends, to the SSD. Finally, many databases allow users to write SQL queries to run as stored procedures [43, 44]. Similarly, many key-value stores, such as Redis [40], Splinter [39] and Comet [21] support extensibility with user-defined functions. POSH, in contrast, focuses just on a locality-aware shell, to enable the extensibility of remote filesystems. POSH pushes computation to proxy servers (that do not require custom hardware), without forcing the user to explicitly decide which operations should be offloaded.

NFS Optimizations and filesystems. NFS, starting in version 4.2 [29], offers support for some server-side operations, such as server-side copy [41]; however, NFS does not have support for offloading arbitrary programs. NFS allows batching operations via compound operations [29]; Juszczak [37] describes techniques to batch writes. vNFS [10] offers a new API for NFS that supports batching and vectorizing filesystem operations. This technique reduces the latency of running commands such as `tar` over NFS, but still requires moving the data across the network, instead of pushing the computation to the data. CA-NFS [6] attempts to improve application performance in a multi-tenant scenario by adaptively scheduling certain client operations to run asynchronously during periods of high system load. POSH currently does not handle multi-tenancy, but could use similar methods to factor system load to adapt scheduling decisions. BlueSky [58] proposes running proxy servers in the cloud that serve data from slower blob storage; these proxy servers can expose NFS access to the data. POSH, in contrast, uses proxy servers to push application code.

Distributed execution engines. Distributed cluster computation systems such as Spark [61], MapReduce [13], Dryad [31], Hadoop [19] also automatically parallelize computation on large datasets, but require that users follow a specific API. Systems such as gg [17] and UCop [15] take "everyday applications," such as software compilation, unit testing and video encoding, and automatically parallelize them in the cloud. These systems focus on compute-intensive workloads, not I/O-intensive, and do not necessarily make decisions about scheduling computation to preserve data locality, unlike POSH.

Code offloading and type systems. Many systems enable *code offloading*, to implement distributed applications on many mobile devices that can benefit from computation on nearby servers [8, 26, 35]. Some of these systems require specific programming language constructs to offload processes and partition programs [8, 27, 34–36]. Recently, Pyxis [11] optimizes database applications by *automatically* offloading procedures to run at the database server. It uses program analysis to determine what to offload, while POSH uses per-command annotations. Split Annotations [46] proposes using per-function annotations to determine how to split and pipeline function calls in data analytics workloads, to enable cross-function cache pipelining and parallelization. These annotations are fundamentally different from POSH's shell annotations: POSH's annotations attempt to understand the command-line semantics of shell commands, which tend to have a much more varied structure than function calls.

Command-line tools. Many command-line tools allow users to automatically parallelize and execute shell commands remotely. `rsh` [3] enables remote execution of shell commands. `pssh` [12] allows users to execute commands over SSH in parallel on other machines. GNU Parallel [56] splits input arguments and executes jobs over these inputs in parallel across one or more machines. POSH also parallelizes commands on remote machines, but *automatically* decides how to offload and schedule commands so users do not need to explicitly program when to offload code. POSH does not rely on SSH access to the storage servers and can be used on top of a service such as NFS, where remote shell access may be prohibited.

3 SYSTEM OVERVIEW

POSH consists of three main components: a shell annotation interface (§4), a parser and scheduler (§5), and an execution engine (§6). This section briefly describes each component and how they link together, pictured in Figure 1.

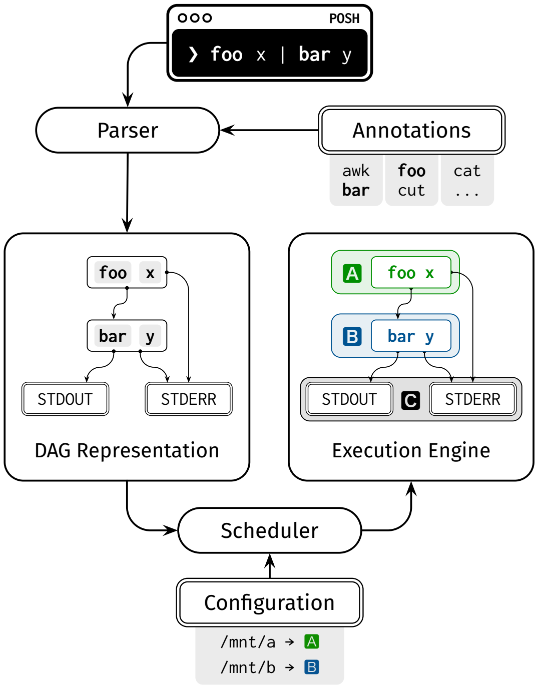


Figure 1: In POSH's main workflow, a shell command is passed to the parser, which uses the annotations to generate and schedule a DAG representation of the command. The DAG includes which machine to run each command on, A, B, or C (client) here. The execution engine finally runs the resulting DAG.

Annotation interface. POSH, on bootup, requires users to provide a file containing a list of annotations for any commands they want POSH to consider offloading. Annotations are written *once per command*, e.g., once for `grep` or once for `awk`, so POSH can then accelerate shell pipelines that combine these commands with standard constructs, such as anonymous pipes. We envision that developers can share annotations for popular programs, so users do not necessarily need to write their own annotations; crowdsourcing annotations has seen success with TypeScript [2, 42, 49].

Parser and scheduler. Given a shell program, the POSH parser turns each pipeline (each line of the program, potentially consisting of several commands combined by pipes and redirects) into a directed acyclic graph (DAG). This graph represents the input-output relationship between commands, the standard I/O streams (`stdin`, `stdout` and `stderr`) and redirection targets, as shown in Figure 1. POSH then parses each individual command and its arguments using the corresponding annotation and completes the DAG by including additional input and output dependencies of the pipeline.

The parser finally runs a scheduling algorithm on the DAG and assigns an execution location to each command in the pipeline. In order to do this, the parser requires extra *configuration information* that specifies a mapping between each mounted client directory and the address for a machine running a proxy server for the corresponding directory (if any).

Execution engine. After POSH has parsed and scheduled a shell pipeline, it executes the command across the underlying execution engine. The execution engine consists of one or more proxy servers, each associated with a specific remote client mount, either at the storage server, or in a nearby node with access to the same data. Additionally, one "proxy server" runs at the client to execute any local computation. POSH ensures that the entire command looks like it has been running locally, even if processes had been offloaded to proxy servers.

4 Shell Annotations

POSH must correctly understand the semantics of shell commands, which can be challenging because of the wide range of syntax allowed by command lines. In this section, we discuss the motivation and design of POSH's shell annotation layer.

4.1 Motivation for Shell Annotations

In order to schedule and execute shell pipelines in a way that minimizes data movement, POSH must understand the semantics of command-line pipelines. Concretely, annotations must reveal enough information that allows POSH to determine:

1. Which commands can be safely offloaded to proxy servers.
2. If any commands in the pipeline filter their input.
3. If any commands can be split in a data-parallel way into multiple processes.

Consider a simple pipeline: `cat A B C D | grep "foo" | tee local_file.txt`. POSH could try to offload any of the three commands: `cat`, `grep`, or `tee`. To determine which commands are safe to offload, POSH must understand which files (if any) `cat`, `grep` and `tee` access, and where these files live. Therefore, POSH must determine which arguments to the three commands represent file paths. However, outside of the program, all of these arguments are seen as generic strings. For example, consider the following four commands:

```
cat A B C D | grep "foo"
tar -cvf output.tar.gz input/
tar -xvf input.tar.gz
git status
```

The `cat` command takes in four input files, while the argument to `grep` is a string. The second command, `tar -cvf`, takes an output file argument followed by `-f`, followed by an input file argument (not preceded by a short option). The third command, also `tar`, takes an input file argument followed by `-f` and implicitly takes its output argument as the current directory. Finally, `git` also implicitly relies on the current directory as a dependency. Without a formal way to model the argument structure for each command, POSH could not determine the file dependencies for each command.

Secondly, in order to produce an execution schedule that reduces data movement, POSH needs to know the relationship between the inputs and outputs of a command. In the `cat | grep` example, if the file argument to `cat` is remote, to minimize data movement, POSH cannot *just* offload the `cat` command. Since `cat` usually produces the same amount of output as input, but `grep` usually filters its input, POSH must also offload `grep`.

Finally, suppose `cat` had multiple file arguments, but these files lived on different mounts (e.g., a pipeline that processes logs from different servers). POSH could not safely offload this command to a single proxy server, as the proxy server may not have access to all the mounts. However, many command-line programs perform map functions over each line in the file in sequence. For example, `cat` prints all lines to the output, and `grep` filters the input line by line. Therefore, these commands can be split into processes across their input files and then offloaded to different proxy servers. However, parallelization is not safe for all commands: `wc`, for example, "reduces" the input. Without a formal model for the command's semantics, POSH could not make optimal scheduling decisions.

4.2 Annotation Interface

Annotations need enough information to allow POSH to deconstruct, parse and schedule each command. Annotations contain two types of information: argument-specific and command-specific information. First, they contain a list of arguments along with a type assignment for each argument. Second, they contain information relevant to parsing the entire command line, either semantic information relevant to scheduling, or custom parsing options. The annotation interface is inspired by the POSIX conventions for command-line arguments and their GNU extensions [23], which are followed by a multitude of UNIX utilities. If a program does not follow these conventions, POSH may not be able to determine how to accelerate it. We describe both parts of the annotation in turn.

4.2.1 Argument-Specific Information

POSH supports the following classes of command-line arguments:

1. A single option with no arguments (e.g., `-d` or `--debug`).
2. An option, followed by one or more parameters (e.g. `-f <file>`).
3. A parameter without a preceding option (e.g. the arguments in `cat A B C D`).

The annotations must specify the following information for each argument that has associated parameters:

1. The **`short`** or **`long`** option name: This is only relevant for arguments preceded by options.
2. The **`type`**: `input_file`, `output_file`, or `string`.
3. The **`size`**: `1`, `specific_size(x)`, or `list` (for variable size).
4. If the argument is **`splittable`**: If an argument has multiple parameters, the **`splittable`** keyword specifies that the command can be split in a data-parallel way across this argument (this is only allowed for up to a single argument).

The above format applies to many popular UNIX commands and core utilities. For example, the annotation for `cat` may look like:

```
cat:
  - PARAMS:
    - type:input_file,splittable,size:list
```

This specifies that `cat` takes in one or more input files, and it can be split across its inputs.

4.2.2 Command-Specific Information

Annotations contain information relevant to the semantics of the entire command, specified by the following keywords:

- **`needs_current_dir`**: Whether the command implicitly relies on the current directory.
- **`splittable_across_input`**: Whether the command can be split *across its standard input*. In the example pipeline, if the `cat` is split into separate `cats`, POSH would also need to split the `grep` command into separate commands to truly take advantage of parallelism.
- **`filters_input`**: Whether the command is *likely* to have a smaller output than input.
- **`long_arg_single_dash`**: Most programs use double dashes before long arguments (e.g. `--debug`), but some programs require long arguments be preceded by a single dash. (e.g. `-debug`).

4.2.3 Annotation Conflicts

Some commands can be invoked with flags whose behavior changes depending on which other flags are present. For example, the annotation for a `tar` invocation used to *create* an archive could be:

```
tar: [filters_input]
  - FLAGS:
    - short:c
    - short:z
  - OPTPARAMS:
    - short:f,type:output_file,size:1
  - PARAMS:
    - type:input_file,size:list
```

However, developers commonly invoke both `tar -x` and `tar -c`, to extract and create a tarball, respectively. The assignment for the `-f` flag conflicts; it would be an `input_file` in the `-x` case, but an `output_file` in the `-c` case. POSH supports this by allowing multiple annotations per command. In particular, a client can include one annotation *per type of invocation* for every binary, and POSH will try all annotations until it finds one that fits the current invocation.

4.3 Correctness and Coverage

Potential mistakes. POSH depends on correct annotations for optimal execution: if the annotations are incorrect in some way, POSH does not make guarantees about the correctness or the performance of the resulting execution. Annotations with incorrect type semantics (assigning an argument with `str` instead of `input_file`) or parallelization semantics (specifying a command is **`splittable`** when the command needs all input files concurrently) could cause execution errors. POSH might schedule the command on a machine without access to a necessary file, or incorrectly try to split the command into parallel workers. Annotations might not include potential optimization information, by omitting that a command filters its input or it can be parallelized. In this case POSH might not make the optimal scheduling decision, but execution will still be correct.

Finally, a command, such as `awk`, could either filter or increase its input depending on its invocation. `awk` could include multiple annotations for each separate program string, which separately specify or omit the **`filters_input`** keyword.

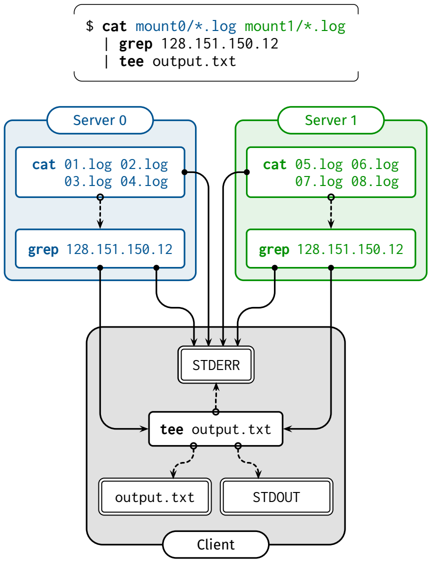
Mitigations. In practice, we expect that a community of developers would maintain and crowdsource a set of "verified" annotations; other annotation-based systems such as Typescript [42] make this assumption. To prevent incorrect type semantics, POSH could use a sandbox on top of the filesystem interface that checks if the program only reads and writes to files specified by the annotation, and crash the execution if the program accesses a file outside of its dependency list [17, 20]. Finally, future work could explore profiling commands to automatically deduce whether command invocations produce less, more or the same amount of input data as output data.

Coverage. The POSH interface covers the command-line syntax for a wide range of command line programs, as specified by the GNU command-line standard syntax and extensions [23]. Along with the source code of POSH, we provide annotations for 21 different commands, which cover a wide range of unmodified shell applications used in our evaluation (§7).

However, POSH will not cover *all* parsing options for all programs. While POSH can interpret wildcards ("*") when listing file paths, it will not do any custom parsing to list paths. For example, `ffmpeg` allows users to provide input files (input frames) based on a pattern, such as "`%04d.jpg`", which corresponds to all files between `0000-9999.jpg`. POSH will not parse commands whose file dependencies are specified *dynamically*, via a pipe: e.g., "`find . -type -iname "*.jpg" | xargs -i mogrify -resize 100x100`" could be used to dynamically list all `jpg` files and resize them with `ImageMagick`. Since POSH does not know what the file inputs to the `mogrify` command are upfront, it cannot decide whether to offload the `mogrify` command or not.

5 POSH's Parser and Scheduler

This section discusses how POSH solves two challenges in executing shell commands efficiently across a set of proxy servers: (1) scheduling pipelines to minimize data movement (§5.2) and (2) correctly parallelizing pipelines (§5.3). We begin by discussing how POSH constructs an intermediate program representation of the command that allows it to



```
$ cat mount0/*.log mount1/*.log
  | grep 128.151.150.12
  | tee output.txt
```

Figure 2: DAG representation of a simple shell program that uses `cat` and `grep` to analyze logs across different mounts. POSH uses its scheduling and parallelization mechanisms to offload the `cat` and `grep` to each server.

effectively solve these two problems.

5.1 POSH's Program Representation

POSH needs a program representation that allows the runtime to see every data source (file that is read), data sink (output file that is written to), and flow (pipe) that connects two commands within the pipeline. Understanding which files commands read and write allows POSH to determine the execution location closest to the data for each command. Understanding the connections in the pipeline allows POSH to see how data flows between commands and allows POSH to preserve dependencies when parallelizing nodes.

POSH represents parsed shell pipelines as directed acyclic graphs (DAGs), which contain *nodes* and *streams*. Nodes represent single commands (command nodes) along with each of their arguments, and parsed annotation metadata (argument types, if it is **`splittable`**, and if it filters its input). Nodes also represent sources and sinks in the entire pipeline: reading from `stdin` (read nodes) or writing to `stdout` or a file (write nodes). Streams represent the data flows in and out of these nodes.

Given a shell pipeline, POSH constructs a DAG that models the dependencies between commands (UNIX pipes) and standard I/O streams (`stdin`, `stdout` and `stderr`). POSH then builds a custom argument parser for each command from the annotations, to determine which arguments actually appear in each command's invocation and what their corresponding types are. For example, if a `tar -c` invocation contained the `-f` argument, POSH knows the string following `-f` is an `output_file`.

Figure 2 shows an example DAG generated for the simple program discussed in §4.1, that runs `cat` and `grep` on files stored in different mounts and pipes the final output to `tee`.

Algorithm 1 POSH Scheduling Algorithm

```
function SCHEDULE(dag)
  for node ∈ dag.GETNODES do
    if CONSTRAINT(node) != NULL then
      node.location ← CONSTRAINT(node)
  for source_node ∈ dag.GETSOURCES do
    path ← GETPATHTOSINK(dag,source_node)
    sink_node ← path[path.length() - 1]
    if source_node.location == sink_node.location then
      for node ∈ path do
        if node.location == NULL then
          node.location ← source_node.location
    else
      $min\_weight \leftarrow 1$
      edges ← { }
      for (node,next) ∈ path do
        if ISFILTERNODE(node) then
          $min\_weight \leftarrow min\_weight/2$
        edges[(node.id,next.id)] = $min\_weight$
      for (node,next) ∈ path do
        if node.location == NULL then
          if edges[(node.id,next.id)] ≥ $min\_weight$ then
            node.location ← source_node.location
          else
            node.location ← sink_node.location
        else if node.location != CONSTRAINT(node) then
          node.location ← CLIENT
```

POSH schedules and parallelizes the workload by first splitting the command into separate `cat` and `grep` commands that run at each proxy server and then merging the outputs at the client. Sections 5.2 and 5.3 discuss these steps in turn.

5.2 Scheduling

POSH's scheduling algorithm seeks to minimize data movement, as it assumes that in scenarios where the computation is I/O bound rather than CPU bound, minimizing data movement will reduce end-to-end latency. However, POSH does not know how much data a command will produce prior to execution. We describe the exact setup of the problem and POSH's greedy algorithm that uses information from POSH's annotations to estimate how to minimize data movement.

Problem setup. The POSH scheduling algorithm, summarized in Algorithm 1, takes the DAG representation of the command discussed in §5.1 and assigns an execution location to each node. POSH must pay attention to two concerns: *constraints* on where certain nodes can execute and the *number of bytes transferred across edges* in the DAG. The first concern arises because certain proxy servers might not be able to execute certain nodes as they do not have access to all the necessary files. The second concern arises because POSH seeks to minimize the number of bytes transferred between two locations across the network. Concretely, consider a `cat` node (operating on a remote file) that pipes its output to a `grep` node, that filters

Algorithm 2 POSH Scheduling Algorithm Helper Functions

```
 1: /* Returns constrained location assignment for node, if any. */
 2: function CONSTRAINT(node)
 3:   loc ← NULL
 4:   if ISREADNODE(node) then
 5:     loc ← GETREADLOC(node)
 6:   else if ISWRITENODE(node) then
 7:     loc ← GETWRITELOC(node)
 8:   else if ISCMDNODE(node) then
 9:     deps ← Set()
10:     for file ∈ GETFILEDEPENDENCIES(node) do
11:       deps.append(GETLOCATION(file))
12:     if deps.length() > 1 then
13:       loc ← CLIENT
14:     else if deps.length() == 1 then
15:       loc ← deps[0]
16:   return cost
17: /* Given a source node, trace a path to the sink via stdout paths. */
18: function GETPATHTOSINK(dag,node)
19:   path ← [node]
20:   while node.children.length() > 0 do
21:     path.append(GETSTDOUT(node))
22:   return path
```

its input, which in turn writes its output to `stdout` on the client. To schedule the least data movement across the network, POSH should offload both the `cat` and `grep` commands: in the path through the `cat`, `grep` and the `stdout` nodes, the minimum cut (edge with least data transferred) occurs after `grep`.

Step 1: Resolving constraints on each node. Algorithm 2 summarizes the helper functions for the scheduling algorithm, including a function, `Constraint`, that determines whether a node in the DAG has any constraints on execution location (lines 1-16). Read and write nodes are assigned to the location of their input or output data streams (lines 4-7). Command nodes that access files on a single mount are greedily assigned to execute on the proxy server corresponding to that mount (lines 14-15). Finally, command nodes that access files from multiple different mounts are always assigned to execute on the client (lines 12-13). Only the client has access to all mounts; by default, proxy servers only serve requests for a single mount. We discuss an optimization to this decision in §5.3, for special cases where the command can be parallelized.

Step 2: Minimizing data transfer. In Algorithm 1, after assigning locations to nodes with constraints (lines 1-4), POSH assigns locations for the remaining command nodes. POSH first iterates through all the source nodes of the graph, traces the path from each source to a sink node with the `GetPathToSink` helper function defined in lines 18-19 of Algorithm 2, and considers each path individually. Each path is guaranteed to be linear because each node in the graph effectively has one child. POSH assumes most data transfer occurs along `stdout` streams: even though command nodes have two children (one edge to `stdout` and one to `stderr`), POSH only pays attention to `stdout` connections. Read nodes can have one child by definition, and write nodes are sinks, so have no children.

Within a path, when the source and sink nodes have the same location, scheduling is simple: all nodes along the path from the source to the sink are assigned that location (lines 8-11). However, when the source and sink have different locations, the scheduler must find the edge along which cross-location data transfer should occur: to minimize data transfer, this should be the edge where the *least* data flows. POSH first iterates along each edge in a path and assigns relative weights according to heuristics (lines 13-18). POSH assumes nodes produce the same amount of output as input, or filters input by half (lines 16-18). The helper function `IsFilterNode` called on line 16 returns true for commands whose annotations indicate that they filter their input from the **`filters_input`** keyword. This heuristic, that filter commands halve their input, obviously does not fit all cases. Some filters, such as `wc`, usually produce much less than half the input. To find the minimum cut, only the relative ordering of edges matters, rather than the absolute weight values.

POSH then iterates along each path and schedules each unassigned node's location to be either the source location or the sink location, depending on if the node is before or after the minimum cut edge. For example, if a path contains `cat`, `grep`, and `cut`, and writes to `stdout` on the client, POSH determines the minimum weight edge is in between `cut` and the `stdout` write node. When there are conflicting assignments from nodes appearing in two different paths, POSH schedules the node on the client (lines 25-26).

5.3 Parallelization

The second challenge POSH resolves is determining when commands are safe to parallelize and then guaranteeing correct execution while parallelizing nodes. After determining a command is safe to split, correctly executing the command in a data-parallel way requires: (1) splitting the command only across the argument that can be split while preserving the other arguments and (2) stitching the outputs of the command back together in the correct order.

Determining which nodes to parallelize. POSH can automatically parallelize nodes that are safe to parallelize across the files they access, or across their input edges. In the first case, to parallelize the command `cat -n A.txt B.txt C.txt`, POSH needs to know that the file arguments are `A.txt, B.txt` and `C.txt`, and that the "-n" flag must be preserved. POSH uses the per-argument **`splittable`** keyword in an annotation, which indicates that the command can be split across a particular argument (here, the file argument). In the above example, POSH would replace the single node for `cat` with three nodes, each with the `-n` flag and one of `A.txt, B.txt` or `C.txt`.

In the second case, POSH allows nodes to also be parallelized *across their input streams*. Concretely, if the `cat` above piped its output to `grep`, there would not be much performance benefit to parallelizing `cat`, unless POSH also split `grep` across the three input `cat` nodes. POSH determines which commands this parallelization is safe for via the **`splittable_across_input`** keyword. The annotation for `grep` would include this keyword, so POSH would replace the `grep` node with three `grep` nodes for each `cat` node. This is not safe for commands that reduce or merge the input such as word count (`wc`): since the annotation for `wc` would not include this keyword, POSH would ensure that the output of the prior commands are merged before executing `wc`.

Splitting across mounts. When POSH splits nodes that can be parallelized, it will inherently split commands that read and write to *different mounts*. This allows the scheduler to bypass the restriction in Algorithm 2, lines 12-13. If a single command accesses files in different mounts, but the command can be split, instead of assigning this node to run at the client, POSH will split it into multiple workers that run in parallel on different proxies. POSH by default parallelizes commands across machines, but within a single machine, the *maximum splitting factor* parameter determines the degree to which to split further. The default value is 1, but increasing the splitting factor to a value $s > 1$ causes POSH to split a command into s commands that each operate on $\frac{n}{s}$ sized chunks of the input files, where n is the number of files that the node accesses on a single machine.

Correctly preserving output order. In order to ensure that the output of the entire pipeline is correct, when POSH splits a command in parallel, it must ensure that any node that reads the output of this node now reads the output of the replacement nodes *in sequence*. When nodes execute, they process their inputs in sequential order, guaranteeing correct output order.

6 POSH CONFIGURATION AND EXECUTION

In this section, we discuss how to configure POSH and how POSH executes programs.

6.1 POSH Configuration

To understand how to setup and configure POSH, consider a client that has access to folders on two NFS servers, within a nearby datacenter. Each NFS administrator has agreed to allow a separate proxy server, that resides within the same datacenter, to access the same mounts on behalf of the client (via NFS).[1] To use POSH, each of the two proxy servers must run the POSH *server* program, which is configured with a list of client credentials mapped to the folder paths each client has access to (which the proxy servers themselves access via NFS). The client must run the POSH *client* program, which takes in a file containing the shell annotations, and a configuration file that specifies a mapping from locally mounted folders, to the addresses for proxy servers for those mounts. In the previous example, the client's configuration file would map each of the two mounted folders to the corresponding proxy server. Together, the client and the two proxy servers make up POSH*'s execution engine*, which can be used to execute any schedules POSH's parser creates.

6.2 Execution Engine

After the client schedules a shell pipeline, the execution engine can execute the DAG.

Setup Phase. First, the client divides the DAG into subgraphs that need to execute on different machines (including a subgraph that will execute on the client itself). The client handles setting up persistent connections with proxy servers for any pipes between DAG nodes assigned to two different machines. Finally, when all pipes are setup, the client sends a request to each proxy server to start executing their portion of the DAG.

Execution Phase. Once each proxy server receives a request to execute a subprogram DAG, it will first spawn all the nodes in the DAG corresponding to processes that need to be executed (e.g., `cat` or `grep`). To redirect I/O between processes, POSH spawns a thread for each redirection that needs to occur and copies the output from each node to the correct pipe, TCP stream or file. For nodes that have multiple inputs, nodes process these inputs sequentially. Nodes that send output to nodes with multiple inputs buffer the content until the receiver starts processing that node's output.

6.3 Implementation

POSH is implemented in about 12,400 lines of Rust code. POSH uses Rust's CLAP library [38] to build custom parsers for each command, based on the command's annotation, to find out which arguments are actually present in an invocation.[2]

7 METHODOLOGY

We evaluate POSH by measuring its performance impact over five unmodified I/O-intensive shell applications. This section describes our evaluation methodology: the applications, why they cover a broad range of I/O-intensive shell applications, and our experimental setups.

7.1 Applications

For each application, in our evaluation setting, we assume that all input data files and intermediate files live on a remote NFS mount, so POSH accelerates these applications by preventing unnecessary data movement. Some applications require writing the final output to `stdout` or a file on the client; we specify this on a per-application basis.

Ray-tracing log analysis. The first application represents a best-case workload for POSH: it is computationally light, can be parallelized, and its output is a tiny fraction of the data it reads. The application analyzes logs of a massively distributed research ray-tracing (computer graphics) system [18], to track a

[1] The remote fileserver itself can also run the POSH server program and act as a proxy server.

[2] The implementation of POSH is available at https://github.com/deeptir18/posh.

task (a simulated ray of light) through the path of workers it traversed. The analysis first cleans and aggregates each worker's log into one file with `cat`, `grep`, `head` and `cut`. It then runs `sed` to search for the path of a single ray (e.g., a straggler) across all the workers and stores the final output on a file at the client.

Thumbnail generation. The next application is CPU-intensive, but still produces output that is a tiny fraction of its input, and is highly parallelizable. The application uses ImageMagick [30] to generate thumbnails of size 10 KB for each image in a folder of about 1090 images, each about 4 MB large; the output thumbnails are also written to the remote mount.

Port scan analysis. The third application is computationally heavy, but not parallelizable, and involves data transfers of large files. ZMap [16] is a network scanning tool that performs Internet-wide scans of the public IPv4 address space. Network security researchers run the following shell application to analyze 40 GB subset of a full Internet scan of port `80`; the final output file is stored in the remote mount.

1. Clean the raw data with a program called `zannotate`.
2. Use a JSON processing tool, `jq`, to isolate the *IP* and *AS ID#* (Autonomous System ID) columns.
3. Use `pr` to merge the columns together.
4. Use `awk` to count the number of IPs per AS.

Distributed log analysis. The next application is a synthetic benchmark that models system administrators running analysis on logs across *different storage servers*, to search for an IP address within the access logs stored across different machines. It is computationally light, highly parallelizable, and the output is a tiny fraction of the input. This workload runs `cat` over all of the files and then filters for a particular IP with `grep` and writes the results back to the file stored locally at the client. Each of five storage servers contains approximately 15 GB of logs from the SEC's EDGAR Log File Data Set [53].

Git workflow. The final application, a `git` pipeline on the Chromium [24] repository, attempts to imitate a developer's `git` workflow and is extremely *metadata-heavy*. After rolling back the repository by 20 commits and saving each commit's patch, the workload successively applies each patch and runs three `git` commands: `git status`, `git add .`, and `git commit -m`.

7.2 Setup and Baselines

Baselines. For all workloads, we compare POSH to two NFS configurations, one with with synchronous operations (`rw,sync,no_subtree_check`) and the other with asynchronous operations (`rw,async,no_subtree_check`).

POSH configuration. For all experiments, unless otherwise specified, the POSH proxy runs on the same machine as the NFS server. The POSH client also mounts the NFS folder locally in case some computations must run on the client. Section 8.2.1 evaluates the impact of placing the proxy directly at the storage server versus on another machine.

Network settings. We focus on two network scenarios:

1. Client and server in the same Google Compute Platform (GCP) region (`us-west2`). The RTT and throughput between the two machines are 0.5 ms and 5-10 Gbps, as measured by `ping` and `iperf`.
2. Client in a university network (at Stanford) and server in a nearby GCP region (`us-west2`). The RTT and throughput between the client and server are approximately 20 ms and 600 Mbps.

Setup. GCP client, proxy, and storage machines are configured with 4 vCPUs, 15 GB of memory, and 10 Gbps egress network bandwidth (`n1-standard-4`); they run Ubuntu 19.04. The client at Stanford runs Ubuntu 18.04. All storage servers store data on regional persistent SSDs.

8 EVALUATION

Our evaluation seeks to answer several questions: *(1)* Can POSH accelerate end-to-end pipelines that use many different command-line tools to access remote data over NFS (§8.1)? *(2)* What is the best way to configure POSH (§8.2)? *(3)* Where do performance benefits come from (§8.3)?

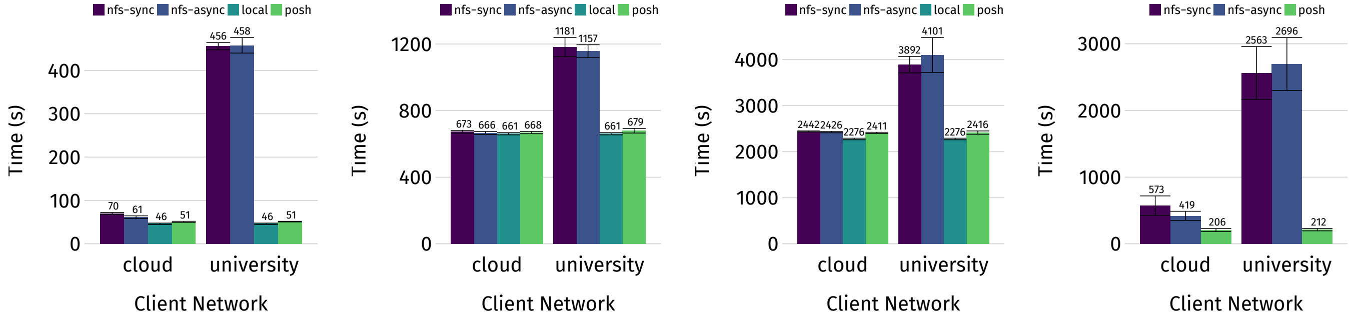
8.1 End-to-End Application Performance

For each workload described in §7.1, Figures 3 and 4 show the performance of POSH compared to `bash` over NFS for two network settings: one where the client is in the same GCP region as the storage server ("cloud") and one where the client is in a university network outside the datacenter ("university"). For `git`, Figure 4 only shows results for a client in the same datacenter.

Summary of all results. On the university-to-cloud network, POSH performs 8× better than `bash` over NFS on the ray-tracing workload, 1-2× better on the thumbnail generation and port scan analysis workloads, and 12.7× better on the distributed log analysis workload. On the cloud-to-cloud network, POSH outperforms `bash` over NFS for the `git` workload and distributed log analysis workload; however, POSH does not outperform `bash` over NFS on the other three applications, partially because these applications are more bandwidth than latency sensitive. We discuss each set of applications below.

Ray-tracing log analysis. This workload sees an 8× improvement on the university-to-cloud network and no improvement on the cloud-to-cloud network. The workload reads 6 GB of input from about 2000 files over NFS, and aggregates them into one 4 GB file, which is written back to NFS. The final output of `sed` on the aggregated file is much smaller (20 lines). POSH prevents 10 GB of data from being copied across the network unnecessarily. On the cloud-to-cloud network, both the overheads of separate filesystem requests (to open and read 2000 files) and the overheads of transferring data to the client are not as large.

Thumbnail generation. This workload sees a 1.7× improvement in the university-to-cloud setting and no improve-



(a) Ray-Tracing Log Analysis **(b)** Thumbnail Generation **(c)** Port Scan Analysis **(d)** Distributed Log Analysis

Figure 3: End to end latency of POSH on four applications, compared to NFS sync, NFS async and local execution time for two networks: one where the client is in a university network and one where the client is in the same GCP region as the storage server. The POSH proxy runs directly on the NFS server. POSH provides between 1.6-12.7× speedups in the university-to-cloud network compared to NFS. Using POSH from a client outside the datacenter results in about same latency as a client inside the datacenter using NFS, with barely any overhead over local execution.

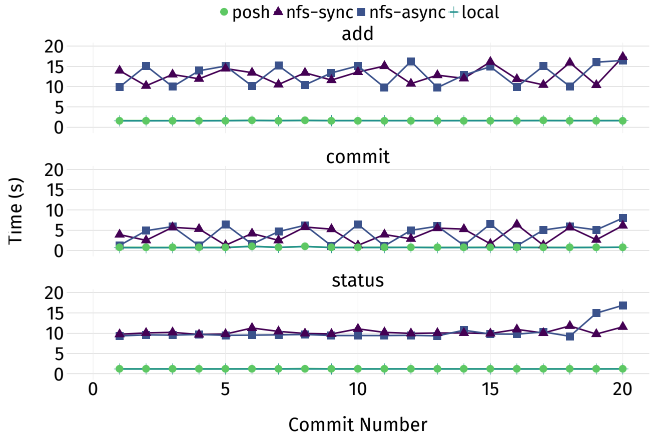


Figure 4: Average latency of 20 `git status`, `git add`, and `git commit` commands run on `Chromium` repo, of POSH compared to NFS and local execution, for a client in the same cloud datacenter as the storage server. POSH provides up to 10-15× speedups by preventing round trips for filesystem metadata calls.

ment in the cloud-to-cloud setting. Generating thumbnails with ImageMagick is computationally heavy: it takes 12 minutes to finish when running locally. While generating thumbnails produces a smaller input (12 MB of thumbnails vs. 15 GB of input images), in the cloud network, transferring 15 GB of data to the client will take about 12 seconds on a 10 Gbps connection. However, in the university-to-cloud setting, POSH attenuates the added delay from transferring data over a slower network.

Port scan analysis. The scanning analysis workload sees a 1.6× benefit with POSH in the university-to-cloud setting, but no benefit in the cloud-to-cloud setting. The scanning workload starts by processing one large 40 GB file with `zannotate` and writing the result back to NFS. This is more bandwidth than latency sensitive, as the application makes fewer filesystem requests across the network than the ray-tracing or thumbnail generation workloads. In addition, `zannotate` is CPU-intensive as it must parse each line of JSON in the input file.

Distributed log analysis. This workload sees a 12.7× improvement in the university-to-cloud setting, because POSH is able to *parallelize* the computation across the five different mounts and only aggregate the result locally. Both offloading the computation in order to prevent data movement as well as running the work on each machine in parallel, instead of sequentially, reduces latency. Even in the cloud-to-cloud setting, this results in a 2× speedup.

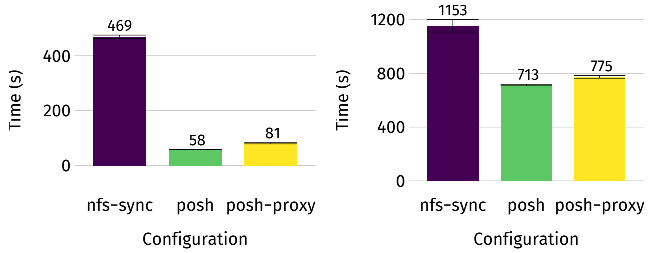
Git workflow. POSH sees the greatest performance benefit, a 10-15× latency improvement, when running `git` commands over NFS. Figure 4 shows the time for each `git status`, `git add`, and `git commit` commands for 20 commits, in the cloud-to-cloud network. We did not perform this full analysis for the university-to-cloud network, because the time to run a single `add` was up to two hours. `git` repositories typically contain many small files; commands like `status` and `add` check the status of every file in the folders to see if it has been modified. This results in NFS making filesystem requests such as `stat` for every file. As a comparison, the ray-tracing log analysis workload makes around 2,500 `open()` calls and 2,500 `stat()` class. For a `git add` in this workload, these numbers are 34,000 and 340,000, respectively, measured by `strace`. By offloading these commands to the server, POSH avoids many unnecessary roundtrips.

8.2 POSH Configuration

We evaluate two aspects of POSH's configuration: placement of proxy servers and maximum parallelization factor within a single machine.

8.2.1 Proxy Placement

Figures 5 and 6 shows the cost of placing the proxy server on a different machine from the storage server, within the same datacenter, for the client at Stanford, for three applications: ray-tracing, thumbnail generation, and `git`. The proxy server is closer to the data than the client; it has both a higher bandwidth



(a) Ray-Tracing Log Analysis **(b)** Thumbnail Generation

Figure 5: Cost of running the POSH proxy on a separate server from the storage server, for a client outside the datacenter ("POSH-proxy", versus POSH where the proxy is at the storage server ("POSH"), and the baseline NFS sync execution time. POSH-proxy has a low overhead over POSH because there is not much overhead to running NFS between two machines in the datacenter.

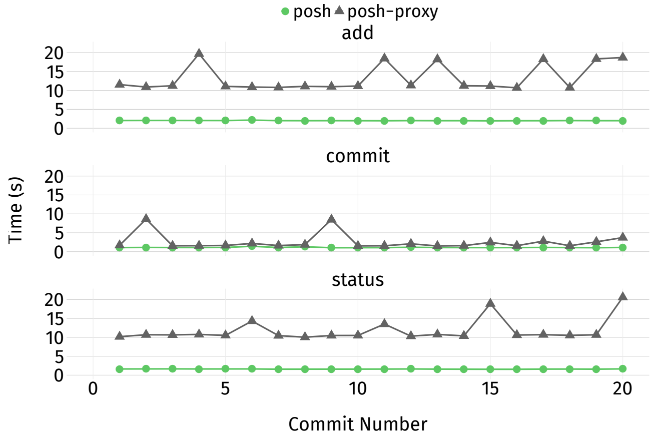
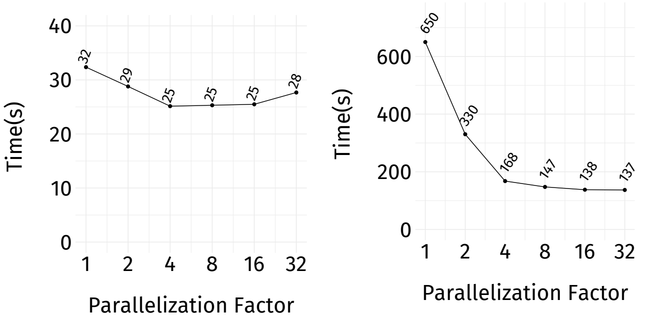


Figure 6: Cost of running the POSH proxy on a separate server from the storage server for a client outside the datacenter for the `git` workload. There is a 10-15x slowdown over POSH running directly at the storage server because running bash over NFS for this workload results in a 10-15x slowdown, due to filesystem metadata calls.

(10 Gbps vs. 600 Mbps) as well as lower latency (0.5 ms vs. 20 ms) connection to the storage server. However, in this setting, the proxy server will have at least the same latency as `bash` over NFS for the cloud-to-cloud setting. The proxy still must perform remote filesystem requests and transfer all the necessary data within the datacenter, because it mounts the data over NFS as well. `bash` over NFS does not have much overhead for the ray-tracing and thumbnail generation workloads, but has a 10-15× overhead for the `git` workload, due to many filesystem metadata calls. However, running `git` at the client outside the datacenter over NFS would have taken on the order of *hours*; POSH merely takes seconds.

8.2.2 Parallelization on a Single Machine

For a 16-core machine, Figure 7 shows the effects of varying the maximum splitting factor for the ray-tracing and thumbnail generation workload. The latency of the ray-tracing workload decreases until $s = 4$ processes and the latency of the thumbnail



(a) Ray-Tracing Parallelization **(b)** Thumbnail Generation Parallelization

Figure 7: Latency improvements from using POSH to parallelize pipelines within a single machine, for the ray tracing and thumbnail generation workloads.

Application	NFS	POSH
Ray-Tracing Log Analysis	10 GB	3 KB
Thumbnail Generation	15 GB	0
Port Scan Analysis	175 GB	0
Distributed Log Analysis	80 GB	76.3 KB

Table 1: Bytes transferred over the network. Data movements are significantly reduced by POSH.

Application	Setup Time
Ray-Tracing Log Analysis	50 ms
Thumbnail Generation	30 ms
Port Scan Analysis	10 ms
Distributed Log Analysis	10 ms
`git status`	0 ms

Table 2: Median setup time.

generation workload decreases until $s = 16$ processes. The limit at 16 is expected for a machine that only has 16 cores. In addition, with higher splitting factors, there is a higher overhead to spawning more threads and context switching. The ray-tracing workload does not benefit from parallelism past 4 threads because it already consists of multiple processes running in parallel, as the workload has many commands (`cat`, `grep`, `cut`, `head`): splitting this further does not provide further benefit.

8.3 Performance Improvements Analysis

Data movement reductions. For each of the log analysis applications and the thumbnail generation application, Table 1 reports the number of bytes of data transferred over the network that this application would generate, as well as the number transferred with POSH and its scheduling mechanism.

POSH overheads. Table 2 reports the latency of POSH's parsing and scheduling steps before execution for a single pipeline (single line of the script) from each application. The overheads are on the order of 10s of milliseconds and barely

Scenario	Latency	Data Movement
NFS Sync	225.8s ± 36.1s	6.38 GB
POSH	221.6s ± 22.1s	6.38 GB
POSH-OPT	33.27s	3.11 GB

Table 3: Expected latency of running POSH with a scheduler that can handle commands that need files from different mounts, for the command `comm A B`, for the university to cloud network.

affect total end-to-end latency. This includes time to parse the configuration file, parse all annotations (which is done once on shell startup), and parse and schedule each command in the pipeline. The ray-tracing and thumbnail generation workloads require resolving more filepaths, causing a larger overhead than the other applications.

9 Limitations and Future Work

Algorithm limitations. POSH's scheduling algorithm handles pipelines that access data on different mounts in a map-reduce style pattern [13], but cannot handle commands which access data on different mounts that cannot be split. Consider a command such as `comm`, that finds the common lines between two files on different mounts: POSH would schedule this command to run at the client, causing no performance benefits over `bash` over NFS. However, if POSH scheduled this command on one of the proxy servers, rather than the client, and transferred the necessary files beforehand, POSH could provide a benefit over NFS. To produce such a schedule, POSH could consider the input files for each command, the data transfer speeds between the proxy machines and the dependencies of the DAG to construct an optimization problem and use standard graph partitioning techniques to solve for the optimal execution location of all nodes. Table 3 shows the expected benefits of a such a scheduler ("POSH-OPT"), for a `comm` command that correlates two 3 GB files stored at two different proxies. It estimates execution time by summing the local execution time of `comm` with the time to transfer one of the files (measured by the time to `scp` the file between the two machines).

Security. POSH allows users to offload parts of shell commands to proxy servers, which could be running directly at the storage layer. POSH currently does not address the security implications of this system design. POSH might allow users to access files such as `/proc/sys` on the storage server which should be restricted. To mitigate this, POSH could ensure that offloaded programs run with limited file access permissions, so they do not access restricted files, and only access files that are specified by the input and output arguments parsed from the annotations. POSH could use sandboxing [20] mechanisms, for example, to restrict users from running offloaded programs that access the network.

Resource management. Since POSH proxy servers could run directly at the storage server, a shared storage server could result in an overloaded CPU and longer latencies for users who expected their commands to take less time since they were offloaded, as well as slowdowns to regular remote filesystem requests. POSH does not currently have policies for load balancing and multitenancy, but could explore policies suggested by prior work [6]. However, initial experiments show that POSH could use a simple policy on the storage server such as monitoring how many cores are in use, and refusing to run programs at the storage server when it is overloaded.

Failure recovery. Currently, POSH does not recover from server-side failures; it does not have mechanisms to migrate or restart jobs if single commands within pipelines fail. Since POSH aims to provide *shell semantics*, which involves streaming data without providing fault tolerance for failed commands, POSH currently does not provide the fault tolerance mechanisms present in standard cluster computing frameworks [17, 31, 61]. However, this is an interesting area of future work: POSH knows exactly what files are being modified or created from the annotations, so POSH could modify programs to write to temporary locations, and only write to the final location when the entire operation is successful.

10 Conclusion

I/O-intensive shell pipelines that run over networked storage incur a significant cost from moving data over the network. We present POSH, a framework that accelerates unmodified shell pipelines with I/O heavy components that access networked storage such as NFS. POSH intercepts shell pipelines and moves individual commands closer to the data by offloading them to run on proxy servers closer to the data than the client. POSH uses metadata about shell commands, written once per command, called annotations, that specify information relevant to safely offloading these commands to proxy servers as well as scheduling them to minimize data movement. POSH uses annotations to schedule and automatically parallelize shell pipelines across the client and proxy servers, while maintaining local execution semantics. We showed that POSH can accelerate a wide range of unmodified shell applications running over NFS and allow them to achieve local execution times.

Acknowledgments

We thank our shepherd, Mahadev Satyanarayanan, and the anonymous ATC reviewers for their invaluable feedback. We are grateful to Shoumik Palkar, Deepak Narayanan, Riad Wahby, Keith Winstein, Liz Izhikevich, Akshay Narayan and members of the Stanford Future Data and SING Research groups for their comments on various versions of this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware, as well as the NSF under CAREER grant CNS-1651570 and Graduate Research Fellowship grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, 1998.

[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[3] BSD Authors. rsh. https://linux.die.net/man/1/rsh.

[4] FFmpeg Authors. FFmpeg. https://ffmpeg.org/.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *HotOS*, 2017.

[6] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. CA-NFS: A congestion-aware network file system. *ACM Transactions on Storage (TOS)*, 2009.

[7] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *Usenix ATC*, 2014.

[8] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP*, 1993.

[9] Ming Chen, Dean Hildebrand, Geoff Kuenning, Soujanya Shankaranarayana, Bharat Singh, and Erez Zadok. Newer is sometimes better: An evaluation of NFSv4.1. In *SIGMETRICS*, 2015.

[10] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Sankar Harihara Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized i/o. In *FAST*, 2017.

[11] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. Speeding up database applications with Pyxis. In *SIGMOD*, 2013.

[12] Brent Chun and Andrew McNabb. pssh. https://code.google.com/archive/p/parallel-ssh/.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[14] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *SIGMOD*, 2013.

[15] John R Douceur, Jeremy Elson, Jon Howell, and Jacob R Lorch. The utility coprocessor: Massively parallel computation from the coffee shop. In *Usenix ATC*, 2010.

[16] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Usenix Security*, 2013.

[17] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Usenix ATC*, 2019.

[18] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. Outsourcing everyday jobs to thousands of cloud functions with gg. *Usenix Login*, 2020.

[19] Apache Software Foundation. Hadoop. http://hadoop.apache.org/.

[20] Tal Garfinkel et al. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[21] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, 2010.

[22] Git SCM. https://git-scm.com/.

[23] GNU. Program argument syntax conventions. https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html, 2020 (Accessed March 28,2020.).

[24] Google. Chromium. https://chromium.googlesource.com/chromium/src, 2020 (Accessed January 4, 2020).

[25] Google. Cloud storage. https://cloud.google.com/storage, 2020 (Accessed May 29, 2020).

[26] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code offload by migrating execution transparently. In *OSDI*, 2012.

[27] Robert S Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Tcl/Tk Workshop*, 1996.

[28] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[29] T. Haynes. NFS Version 4 Minor Version 2. https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion2-41, 2016.

[30] ImageMagick – convert, edit, or compose bitmap images. https://imagemagick.org/.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[32] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *VLDB*, 2017.

[33] Jeroen Janssens. *Data Science at the Command Line: Facing the Future with Time-Tested Tools*. O'Reilly Media, Inc., 1st edition, 2014.

[34] Rakesh Jha, Dennis T. Cornhill, and J. Michael Kamrad. Ada program partitioning language: A notion for distributing ada programs. *IEEE Trans. Softw. Eng.*, 1989.

[35] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *SOSP*, 1995.

[36] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *SOSP*, 1987.

[37] Chet Juszczak et al. Improving the write performance of an NFS server. In *USENIX Winter*, 1994.

[38] Kevin K. Command line argument parser. https://crates.io/crates/clap, 2019.

[39] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *OSDI*, 2018.

[40] Redis Labs. Redis. https://redis.io/.

[41] James Lentine, Anshul Madan, and Trond Myklebust. Accelerating nfs with server-side copy. In *FAST*, 2011.

[42] Microsoft. TypeScript. https://www.typescriptlang.org/.

[43] Microsoft. CREATE PROCEDURE (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver15, 2017.

[44] Oracle. Developing and using stored procedures. https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm, 2020 (accessed May 27, 2020).

[45] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *ISCA*, 1998.

[46] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *SOSP*, 2019.

[47] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 1997.

[48] Google Cloud Platform. A user-space file system for interacting with Google Cloud Storage. https://github.com/GoogleCloudPlatform/gcsfuse, 2020 (Accessed May 29, 2020).

[49] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, 2015.

[50] s3fs fuse. FUSE-based file system backed by amazon S3. https://github.com/s3fs-fuse/s3fs-fuse, 2020 (Accessed May 29, 2020).

[51] s3ql. A full featured file system for online data storage. https://github.com/s3ql/s3ql, 2020 (Accessed May 29, 2020).

[52] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *EuroSys*, 2020.

[53] U.S. Securities, Division of Economic Exchange Commission, and Risk Analysis. Edgar log file data set.

[54] Amazon Web Services. Amazon S3. https://aws.amazon.com/s3/, 2020 (Accessed May 29, 2020).

[55] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *OSDI*, 2014.

[56] Ole Tange. GNU Parallel 2018. https://doi.org/10.5281/zenodo.1146014, March 2018.

[57] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new NAS benchmarks. In *FAST*, 2013.

[58] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. BlueSky: A cloud-backed file system for the enterprise. In *FAST*, 2012.

[59] Louis Woods, Jens Teubner, and Gustavo Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In *SIGMOD*, 2013.

[60] N Yezhkova, L Conner, R Villars, and B Woo. Worldwide enterprise storage systems 2010–2014 forecast: recovery, efficiency, and digitization shaping customer requirements for storage systems. *IDC, May*, 2010.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures

Feng Zhang[1], Lin Yang[1], Shuhao Zhang[2,3], Bingsheng He[3], Wei Lu[1], Xiaoyong Du[1]
[1]*Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China*
[2]*DIMA, Technische Universität Berlin*
[3]*School of Computing, National University of Singapore*
fengzhang@ruc.edu.cn, yanglin2330@ruc.edu.cn, shuhao.zhang@tu-berlin.de, hebs@comp.nus.edu.sg, lu-wei@ruc.edu.cn, duyong@ruc.edu.cn

Abstract

Accelerating SQL queries on stream processing by utilizing heterogeneous coprocessors, such as GPUs, has shown to be an effective approach. Most works show that heterogeneous coprocessors bring significant performance improvement because of their high parallelism and computation capacity. However, the discrete memory architectures with relatively low PCI-e bandwidth and high latency have dragged down the benefits of heterogeneous coprocessors. Recently, hardware vendors propose CPU-GPU integrated architectures that integrate CPU and GPU on the same chip. This integration provides new opportunities for fine-grained cooperation between CPU and GPU for optimizing SQL queries on stream processing. In this paper, we propose a data stream system, called FineStream, for efficient window-based stream processing on integrated architectures. Particularly, FineStream performs fine-grained workload scheduling between CPU and GPU to take advantage of both architectures, and it also provides efficient mechanism for handling dynamic stream queries. Our experimental results show that 1) on integrated architectures, FineStream achieves an average 52% throughput improvement and 36% lower latency over the state-of-the-art stream processing engine; 2) compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4x price-throughput ratio, 1.8x energy efficiency, and can enjoy lower latency benefits.

1 Introduction

Optimizing the performance of stream processing systems has been a hot research topic due to the rigid requirement on the event processing latency and throughput. Stream processing on GPUs has been shown to be an effective method to improve its performance [23, 33, 34, 41, 54, 62, 67]. GPUs consist of a large amount of lightweight computing cores, which are naturally suitable for data-parallel stream processing. GPUs are often used as coprocessors that are connected to CPUs through PCI-e [42]. Under such discrete architectures, stream data need to be copied from the main memory to GPU memory via PCI-e before GPU processing, but the low bandwidth of PCI-e limits the performance of stream processing on GPUs. Hence, stream processing on GPUs needs to be carefully designed to hide the PCI-e overhead. For example, prior works have explored pipelining the computation and communication to hide the PCI-e transmission cost [34, 54].

Despite of various studies in previous stream processing engines on general-purpose applications [5, 23, 25, 33, 54, 62], relatively few studies focus on SQL-based relational stream processing. Supporting relational stream processing involves additional complexities, such as how to support window-based query semantics and how to utilize the parallelism with a small window or slide size efficiently. Existing engines, such as Spark Streaming [57], struggle to support small window and slide sizes, while the state-of-the-art window-based query engine, Saber [34], adopts a *bulk-synchronous parallel model* [66] for hiding PCI-e transmission overhead.

In recent years, hardware vendors have released integrated architectures, which completely remove PCI-e overhead. We have seen CPU-GPU integrated architectures such as NVIDIA Denver [13], AMD Kaveri [15], and Intel Skylake [27]. They fuse CPUs and GPUs on the same chip, and let both CPUs and GPUs share the same memory, thus avoiding the PCI-e data transmission. Such integration poses new opportunities for window-based streaming SQL queries from both hardware and software perspectives.

First, different from the separate memory hierarchy of discrete CPU-GPU architectures, the integrated architectures provide unified physical memory. The input stream data can be processed in the same memory for both CPUs and GPUs, which eliminates the data transmission between two memory hierarchies, thus eliminating the data copy via PCI-e.

Second, the integrated architecture makes it possible for processing dynamic relational workloads via fine-grained cooperations between CPUs and GPUs. A streaming query can consist of multiple operators with varying performance features on different processors. Furthermore, stream processing often involves dynamic input workload, which affects operator performance behaviors as well. We can place operators on different devices with proper workloads in a fine-grained manner, without worrying about transmission overhead between CPUs and GPUs.

Based on the above analysis, we argue that stream processing on integrated architectures can have much more desirable properties than that on discrete CPU-GPU architectures. To fully exploit the benefits of integrated architectures for stream processing, we propose a fine-grained stream processing framework, called FineStream. Specifically, we propose the following key techniques. First, a performance model is proposed considering both operator topologies and different architecture characteristics of integrated architectures. Second, a light-weight scheduler is developed to efficiently assign the operators of a query plan to different processors. Third, online profiling with computing resource and topology adjustment are involved for dynamic workloads.

We evaluate FineStream on two platforms, AMD A10-7850K, and Ryzen 5 2400G. Experiments show that FineStream achieves 52% throughput improvement and 36% lower latency over the state-of-the-art CPU-GPU stream processing engine on the integrated architecture. Compared to the best single processor throughput, it achieves 88% performance improvement.

We also compare stream processing on integrated architectures with that on discrete CPU-GPU architectures. Our evaluation shows that FineStream on integrated architectures achieves 10.4x price-throughput ratio, and 1.8x energy efficiency. Under certain circumstances, it is able to achieve lower processing latency, compared to the state-of-the-art execution on discrete architectures. This further validates the large potential of exploring the integrated architectures for data stream processing.

Overall, we make the following contributions:

- We propose the first fine-grained window-based relational stream processing framework that takes the advantages of the special features of integrated architectures.
- We develop lightweight query plan adaptations for handling dynamic workloads with the performance model that considers both the operator and architecture characteristics.
- We evaluate FineStream on a set of stream queries to demonstrate the performance benefits over current approaches.

2 Background

2.1 Integrated Architecture

We show an architectural overview of the CPU-GPU integrated architecture in Figure 1. The integrated architecture consists of a CPU, a GPU, a shared memory management unit, and system DRAM. CPUs and GPUs have their own caches. Some models of integrated architectures, such as Intel Haswell i7-4770R processor [3], integrate a shared last level cache for both CPUs and GPUs. The shared memory management unit is responsible for scheduling accesses to system DRAM by different devices. Compared to the discrete CPU-GPU architecture, both CPUs and GPUs are integrated on the same chip. The most attractive feature of such integration is the shared main memory which is available to both devices. With the shared main memory, CPUs and GPUs can have more opportunities for fine-grained cooperation. The most commonly used programming model for integrated architectures is OpenCL [49], which regards the CPU and the GPU as *devices*. Each device consists of several *compute units* (CUs), which are the CPU and GPU cores in Figure 1.

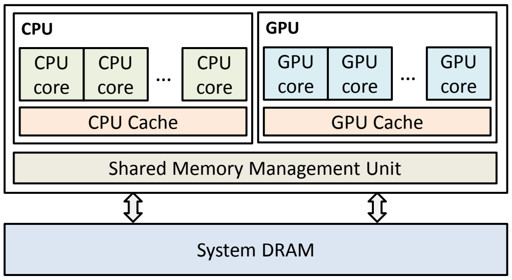


Figure 1: A general overview of the integrated architecture.

We show a comparison between the integrated and discrete architectures (discrete GPUs) in Table 1. These architectures are used in our evaluation (Section 7). Although the integrated architectures have lower computation capacity than the discrete architectures currently, the integrated architecture is a potential trend for a future generation of processors. Hardware vendors, including AMD [15], Intel [27] and NVIDIA [13], all release their integrated architectures. Moreover, future integrated architectures can be much more powerful, even can be a competitive building block for exascale HPC systems [47,55], and the insights and methods in this paper still can be applied. Besides, the integrated architectures are attractive due to their efficient power consumption [15,60] and low price [31].

Table 1: Integrated architectures vs. discrete architectures.

	Integrated Architectures		Discrete Architectures	
Architecture	A10-7850K	Ryzen5 2400G	GTX 1080Ti	V100
# cores	512+4	704+4	3584	5120
TFLOPS	0.9	1.7	11.3	14.1
bandwidth (GB/s)	25.6	38.4	484.4	900
price ($)	209	169	1100	8999
TDP (W)	95	65	250	300

The number of cores for each integrated architecture includes four CPU cores. For discrete architectures, we only show the GPU device.

2.2 Stream Processing with SQL

Although various heterogeneous stream processing systems have appeared [23, 33, 34, 41, 54, 62, 67], we find that most of these systems are used to process unstructured data, and only one work, Saber [34], is developed for structured stream processing on GPUs. Saber supports structured query language (SQL) on stream data [6]. The benefits of supporting SQL come from two aspects. First, with SQL, users can use familiar SQL commands to access the required records, which makes the system easy to use. Second, supporting SQL eliminates the tedious programming operations about how to reach a required record, which greatly expands the flexibility of its usage. Based on the analysis, this work explores stream processing with SQL on integrated architectures.

We consider supporting the basic SQL functions with stream processing, as shown in Figure 2. According to [6], SQL on stream processing consists of the following four major concepts: 1) **Data stream** S, which is a sequence of tuples, $< t_1,t_2,...>$, where t_i is a tuple. A tuple is a finite ordered list of elements, and each tuple has a timestamp. 2) **Window** w, which refers to a finite sequence of tuples, which is the data unit to be processed in a query. The window in stream has two features: *window size* and *window slide*. *Window size* represents the size of the data unit to be processed, and *window slide* denotes the sliding distance between two adjacent windows. 3) **Operators**, which are the minimum processing units for the data in a window. In this work, we support common relational operators including *projection*, *selection*, *aggregation*, *group-by*, and *join*. 4) **Queries**, which are a form of data processing, each of which consists of at least one operator and is based on windows. Additionally, note that in real stream processing systems such as Saber [34], data are processed in *batch* granularity, instead of window granularity. A batch can be a group of windows when the window size is small, or a part of a window when the window size is extremely large.

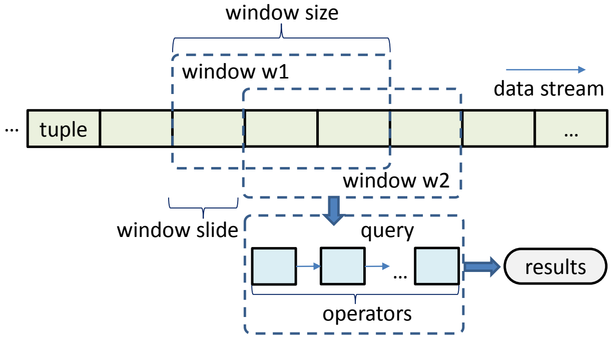


Figure 2: Stream processing with SQL.

3 Revisiting Stream Processing

We discuss the new opportunities (Section 3.1) and challenges (Section 3.2) for stream processing on integrated architectures in this section, which motivate this work.

3.1 Varying Operator-Device Preference

Opportunities: Due to the elimination of transmission cost between CPU and GPU devices on integrated architectures, we can assign operators to CPU and GPU devices in a fine-grained manner according to their device-preference.

We analyze the operators in a query, and find that different operators show various device preferences on integrated architectures. Some operators achieve higher performance on the CPU device, and others have higher performance on the GPU device. We use a simple query of *group-by* and *aggregation* on the integrated architecture for illustration, as shown in Figure 3. The GPU queue is used to sequentially execute the queries on the GPU, while the CPU queue is used to execute the related queries on the CPU. The window size is 256 tuples and the window slide is 64. Each batch contains 64,000 tuples, and each tuple is 32 bytes. The input data are synthetically generated, which is described in Section 7.1. When the query runs on the CPU, *group-by* takes about 18.2 ms and *aggregation* takes about 5.2 ms. However, when the query runs on the GPU, *group-by* takes about 6.7 ms and *aggregation* takes about 5.8 ms.

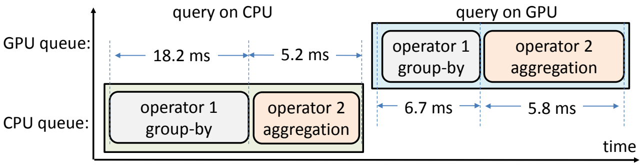


Figure 3: An example of operator-device preference.

We further evaluate the performance of operators on a single device in Table 2. Table 2 shows that using a single type of device *cannot* achieve the optimal performance for all operators. The *aggregation* includes the operators of *sum*, *count*, and *average*, and they have similar performance. We use *sum* as a representative for *aggregation*. From Table 2, we can see that *projection*, *selection*, and *group-by* achieve better performance on the GPU than on the CPU, while *aggregation* and *join* achieve better performance on the CPU than on the GPU. Additionally, *projection* shows similar performance on CPU and GPU devices. Specifically, for *join*, the CPU performance is about 6x the GPU performance. Such different device preferences inspire us to perform fine-grained stream processing on integrated architectures.

Integrated architectures eliminate data transmission cost between CPU and GPU devices. This provides opportunities for stream processing with operator-level fine-grained placement. The operators that can fully utilize the GPU capacity exhibit higher performance on GPUs than on CPUs, so these operators shall be executed on GPUs. In contrast, the operators with low parallelism shall be executed on CPUs. Please note that such fine-grained cooperations is inefficient on dis-

Table 2: Performance (tuples/s) of operators on the CPU and the GPU of the integrated architecture.

Operator	CPU only (10^6)	GPU only (10^6)	Device choice
projection	14.2	**14.3**	GPU
selection	13.1	**14.1**	GPU
aggregation	**14.7**	13.5	CPU
group-by	8.1	**12.4**	GPU
join	**0.7**	0.1	CPU

crete CPU-GPU architectures due to transmission overhead. For example, Saber [34], one of the state-of-the-art stream processing engines utilizing the discrete CPU-GPU architectures, is designed aiming to hide PCI-e overhead. It adopts a bulk-synchronous parallel model, where all operators of a query are scheduled to one processor to process a micro-batch of data [53].

3.2 Fine-Grained Stream Processing

Challenges: A fine-grained stream processing that considers both architecture characteristics and operator preference shall have better performance, but this involves several challenges, from both application and architecture perspectives.

Based on the analysis, we argue that stream processing on integrated architectures can have much desirable properties than that on discrete CPU-GPU architectures. Particularly, this work introduces a concept of fine-grained stream processing: co-running the operators to utilize the shared memory on integrated architectures, and dispatching the operators on devices with both architecture characteristics and operator features considered.

However, enabling fine-grained stream processing on integrated architectures is complicated by the features of SQL stream processing and integrated architectures. We summarize three major challenges as follows.

Challenge 1: Application topology combined with architectural characteristics. Application topology in stream processing refers to the organization and execution order of the operators in a SQL query. First, the relation among operators could be more complicated in practice. The operators may be represented as a directed acyclic graph (DAG), instead of a chain, which contains more parallel acceleration opportunities. Second, with architectural characteristics considered, such as the CPU and GPU architectural differences, the topology with computing resource distribution becomes very complex. In such situations, how to perform fine-grained operator placement for application topology on different devices of integrated architectures becomes a challenge. Third, to assist effective scheduling decisions, a performance model is needed to predict the benefits from various perspectives.

Challenge 2: SQL query plan optimization with shared main memory. First, a SQL query in stream processing can consist of many operators, and the execution plan of these operators may cause different bandwidth pressures and device preferences. Second, in many cases, independent operators may not consume all the memory bandwidth, but co-running them together could exceed the bandwidth limit. We need to analyze the memory bandwidth requirement of co-running. Third, CPUs and GPUs have different preferred memory access patterns. Current methods [5,23,25,33,34,54,62] do not consider these complex situations of shared main memory in integrated architectures.

Challenge 3: Adjustment for dynamic workload. During stream processing, stream data are changing dynamically in distributions and arrival speeds, which is challenging to adapt. First, workload change detection and computing resource adjustment need to be done in a lightweight manner, and they are critical to performance. Second, the query plan may also need to be updated adaptively, because the operator placement strategy based on the initial state may not be suitable when the workload changes. Third, during adaptation, online stream processing needs to be served efficiently. Resource adjustment and query plan adaptation on the fly may incur runtime overhead, because we need to adjust not only the operators in the DAG but also the hardware computing resources to each operator. Additionally, the adjustment among different streams also needs to be considered.

4 FineStream Overview

We propose a framework, called FineStream, for fine-grained stream processing on integrated architectures. The overview of FineStream is shown in Figure 4. FineStream consists of three major components, including performance model, online profiling, and dispatcher. The online profiling module is used to analyze input batches and queries for useful information, which is then fed into the performance model. The performance model module uses the collected data to build models for queries with both CPUs and GPUs to assist operator dispatching. The dispatcher module assigns stream data to operators with proper processors according to the performance model on the fly.

Next, we discuss the ideas in FineStream, including its workflow, query plan generation, processing granularity, operator mapping, and solutions to the challenges mentioned in Section 3.2.

Workflow. The workflow of FineStream is as follows. When the engine starts, it first processes several batches using only the CPUs or the GPUs to gather useful data. Second, based on these data, it builds a performance model for operators of a query on different devices. Third, after the performance model is built, the dispatcher starts to work, and the fine-grained stream processing begins. Each operator shall be assigned to the cores of the CPU or the GPU for parallel execution. Additionally, the workload could be dynamic.

For dynamic workload, query plan adjustment and resource reallocation need to be conducted.

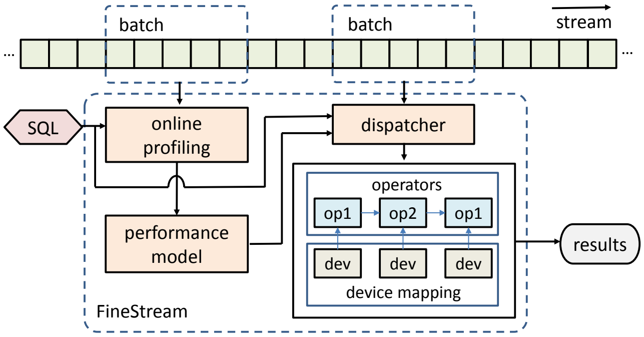


Figure 4: FineStream overview.

Topology. The query plan can be represented as a DAG. In this paper, we concentrate on relational queries. We show an example in Figure 5, where OP_i represents an operator. *OP7* and *OP11* can represent *joins*. We follow the terminology in compiler domain [52], and call the operators from the beginning or the operator after *join* to the operator that merges with another operator as a *branch*. Hence, the query plan is also a branch DAG. For example, the operators of *OP1*, *OP2*, and *OP3* form a branch in Figure 5. The main reason we use the branch concept is for parallelism: operators within a branch can be evaluated in a pipeline, and different branches can be executed in parallel, which shall be detailed in Section 5. The execution time in processing one batch is equal to the traversal time from the beginning to the end of the DAG. Because branches can be processed in parallel, the *branch* with the longest execution time dominates the execution time. We call the operator path that determines the total execution time as $path_{critical}$, so the branch with the longest execution time belongs to $path_{critical}$. For example, we assume that *branch2* has the longest execution time among the branches, its time is $t_{branch2}$, and the execution time for OP_i is t_{OP_i}. *OP7* and *OP11* can also be regarded as branches. Only when the outcomes of *OP3* and *OP6* are available, then *OP7* can proceed. So do to the operators of *OP7* and *OP10* to *OP11*. Assuming OP7 and OP11 are blocking join operators, the total execution time for this query is the sum of $t_{branch2}$, t_{OP7}, and t_{OP11}.

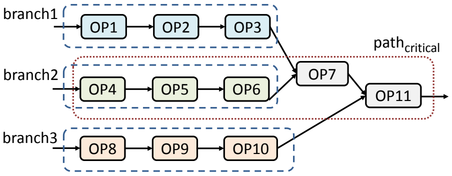


Figure 5: An example of query operators in DAG representation, where OP_i represents an operator.

Operator Mapping. The fine-grained scheduling lies in how to map the operators to the limited resources on integrated architectures. In FineStream, we allow an operator to use one or several cores of the CPU or the GPU device. When the platform cannot provide enough resources for all the operators, some operators may share the same compute units. For example, in Figure 5, *OP1* and *OP2* can share the same CPU cores. If so, the input batches sequentially goes through *OP1* and *OP2* and no pipeline exists between two batches for *OP1* and *OP2*.

Solutions to Challenges. FineStream addresses all the challenges mentioned in Section 3.2. For the first challenge, the performance model module estimates the overall performance with the help of the online profiling module by sampling on a small number of batches, and the dispatcher dynamically puts the operators on the preferred devices. For the second challenge, we have considered the bandwidth factor when building the performance model, which can be used to guide the parallelism for operators with limited bandwidth considered. For the third challenge, the online profiling checks both the stream and the operators to measure the data ingestion rate, and FineStream responses to these situations with different strategies based on the analysis for dynamic workloads. Next, we show the details of our system design.

5 Model for Parallelism Utilization

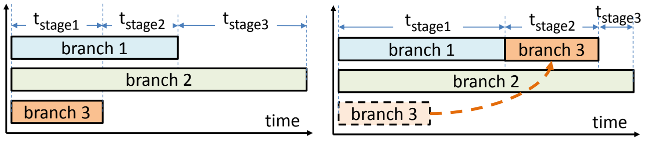
Guideline: A performance model is necessary for operator placement in FineStream, especially for the complicated operator relations in the DAG structure. The overhead of building fine-grained performance model for a query is limited because the placement strategy from the model can be reused for the continuous stream data.

We model the performance of executing a query in this section. The operators of the input query are organized as a DAG. In the performance model, we consider two kinds of parallelism. First, for intra-batch parallelism, we consider *branch co-running*, which means co-running operators in processing one batch. Second, for inter-batch parallelism, we consider *batch pipeline*, which means processing different batches in pipelines.

5.1 Branch Co-Running

Independent branches can be executed in parallel. With limited computation resources and bandwidth, we build a model for branch co-running behaviors in this part. We use *Bmax* to denote the maximum bandwidth the platform can provide. If the sum of bandwidth utilization from different parallel branches, *Bsum*, exceeds *Bmax*, we assume that the throughput degrades proportionally to the *Bmax*/*Bsum* of the throughput with enough bandwidth [60]. To measure the bandwidth utilization, generally, for *n* co-running tasks, we have *n* co-running stages, because tasks complete one by one. When multiple tasks finish at the same time, the number of stages decreases accordingly.

We use the example in Figure 5 for illustration. Assume that the time for different *branches* is shown in Figure 6 (a). If we co-run the three branches simultaneously, then the execution can be partitioned into three stages with different overlapping situations. We use t_{stage1}, t_{stage2}, and t_{stage3} to represent the related stage time when the system has enough bandwidth. Then, if the required bandwidth for $stage_i$ exceeds $Bmax$, the related real execution time t_{stage_i}’ also extends accordingly. We define t_{stage_i}’ in Equation 1. When the platform can provide the required bandwidth, r_i is equal to one. Otherwise, r_i is the ratio of the required bandwidth divided by $Bmax$.



(a) Branch parallelism.

(b) Branch scheduling optimization.

Figure 6: An example of branch parallelism and optimization.

$$t_{stage\_i}{}' = r_i \cdot t_{stage\_i} \tag{1}$$

To estimate the time for processing a batch in the critical path, generally for the branch DAG, we perform topology sort to organize the branches into different layers, and then we co-run branches on layer granularity. In each layer, we perform the above branch co-running. Then, the total execution time is the sum of time of all layers, as shown in Equation 2.

$$t_{total} = \sum_{j=0}^{n_{layer}} \sum_{i=0}^{n_{stage}} t_{stage\_i,layer\_j}{}' \tag{2}$$

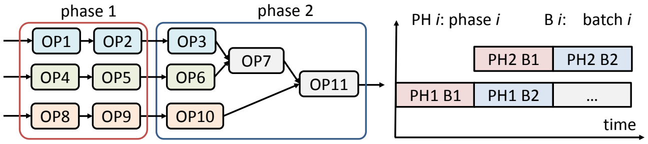
The throughput is the number of tuples divided by the execution time. Assume the number of tuples in a batch is m, then, the throughput is shown in Equation 3.

$$throughput_{branchCoRun} = \frac{m}{t_{total}} \tag{3}$$

Optimization. We can perform branch scheduling for optimization, which has two major benefits. First, by moving *branches* from the stage with fully occupied bandwidth utilization to the stage with surplus bandwidth, the bandwidth can be better utilized. For example, in Figure 6 (b), assume that in $stage_1$, the required bandwidth exceeds $Bmax$, but the sum of the required bandwidth of $branch2$ and $branch3$ is lower than $Bmax$, then we can move the execution of *branch3* after the execution of *branch1* for better bandwidth utilization. Second, the system may not have enough computation resources for all branches so that we can reschedule branches for better computation resource utilization. In *stage1* of Figure 6 (a), when the platform cannot provide enough computing resources for all the three branches, we can perform the scheduling in Figure 6 (b). Additionally, We can perform batch pipeline between operators in each branch, which shall be discussed next.

5.2 Batch Pipeline

We can also partition the DAG into phases, and perform co-running in pipeline between phases for processing different batches. For simplicity, in this part, we assume that the number of phases in the DAG is two. Please note that when the platform has enough resources, the pipeline for operators can be deeper. We show a simple example in Figure 7 (a). The operators in *phase1* and the operators in *phase2* need to be mapped into different compute units, so that these two phases can co-run in the pipeline. Figure 7 (b) shows the execution flow in pipeline. When FineStream completes the processing for *batch1* in *phase1* and starts to process *batch1* in *phase2*, FineStream can start to process *batch2* in *phase1* simultaneously. *Phase1* and *phase2* can co-run because they rely on different compute units.



(a) Phase partitioning.

(b) Batch pipeline.

Figure 7: An example of partitioning phases for batch pipeline.

We need to estimate the bandwidth of two overlapping phases, so that we can further estimate the batch pipeline throughput. The time for a phase, t_{phase_i}, is the sum of the execution time of the operators in the phase for processing a batch. We use t_{phase1} to denote the time for *phase1* while t_{phase2} for *phase2*. When two batches are being processed in different phases in the engine, FineStream tries to maximize the overlapping of t_{phase1} and t_{phase2} of the two batches. However, the overlapping can be affected by memory bandwidth utilization. The online profiling in Section 6.3 collects the size of memory accesses $s_{i,dev}$ (including read and write) and the execution time $t_{i,dev}$ for each operator. The bandwidth of the two overlapping phases is described as Equation 4.

$$bandwidth_{overlap} = MIN(Bmax, \frac{\sum_{i=0}^{m} s_{i,dev}}{t_{phase1}} + \frac{\sum_{i=m+1}^{n} s_{i,dev}}{t_{phase2}}) \tag{4}$$

When $bandwidth_{overlap}$ does not reach $Bmax$, the execution time for processing n batches, $t_{nBatches}$, is shown in Equation 5.

$$t_{nBatches} = n \cdot MAX(t_{phase1}, t_{phase2}) + MIN(t_{phase1}, t_{phase2}) \tag{5}$$

When $bandwidth_{overlap}$ reaches $Bmax$, the execution time of co-running two phases in the pipeline on different batches is longer than any of their independent execution time. We assume that the independent execution time of the longer phase is t_l and the independent time for the shorter phase is t_s. Then, the overlapping ratio for the two phases r_{olp} is t_s divided by t_l. Assuming the total size of the memory accesses for the longer phase is s_l, and the total size for the shorter phase is s_s, then the execution time of the overlapping interval, t_{olp}, is shown in Equation 6.

$$t_{olp} = \frac{s_s + r_{olp} \cdot s_l}{bandwidth_{overlap}} \tag{6}$$

To estimate the time of the rest part in the longer phase, we assume that the bandwidth of the independent execution of the longer phase is $bandwidth_l$. Then, the execution time t_{rest} is shown in Equation 7.

$$t_{rest} = \frac{(1 - r_{olp}) \cdot s_l}{bandwidth_l} \tag{7}$$

Then, when bandwidth $Bmax$ is reached, the execution time $t_{nBatches}$ to process n batches is shown in Equation 8.

$$t_{nBatches} = n \cdot (t_{olp} + t_{rest}) \tag{8}$$

We assume that a batch contains m tuples, and then, the throughput can be expressed by Equation 9. When bandwidth is sufficient, $t_{nBatches}$ is described as Equation 5; otherwise, Equation 8.

$$throughput_{batchPipeline} = \frac{m \cdot n}{t_{nBatches}} \tag{9}$$

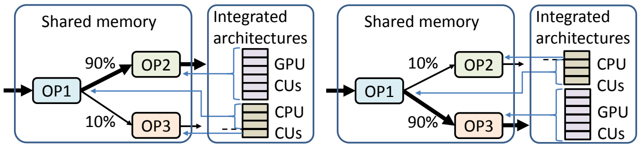
Optimization. Branch co-running can also be conducted in batch pipeline. For example, in Figure 7, the branches in *phase1* can be corun when the system can provide sufficient computing resources and bandwidth. The only thing we need to do is to integrate the branch co-running technique in the potential phases.

5.3 Handling Dynamic Workload

In branch co-running, the hardware resource binded to each branch is based on the characteristics of both the operator and the workload. During workload migration, the workload pressure for each branch may be different from the original state. Hence, the static computing resource allocation may not be suitable for dynamic workload.

A possible solution is to redistribute computing resources to operators in each branch according to the performance model. However, this solution has the following two drawbacks. First, only adjusting the hardware resources on different branches may not be able to maintain the performance, because query plan topology may not fit the current streaming application. In such cases, the query plan needs to be reoptimized for system performance. Second, resource redistribution incurs overhead. Therefore, efficient *resource reallocation* and *query plan adjustment* are necessary for FineStream handling dynamic workload.

Light-Weight Resource Reallocation. In FineStream, we use a light-weight dynamic resource reallocation strategy. When the workload ingestion rate of a branch decreases, we can calculate the reduced ratio, and assume that such proportion of computing resources in that branch can be transferred to the other branches. We use an example in Figure 8 for illustration. In Figure 8 (a), 90% workload after operator *OP1* flow to *OP2*. When the workload state changes to the state in Figure 8 (b), part of the computing resource associated with *OP2* shall be assigned to *OP3* accordingly.



(a) 90% workload goes to OP2.

(b) 90% workload goes to OP3.

Figure 8: An example of adjustment for dynamic workload.

In detail, for the ingestion-rate-falling branch (data arrival rate of this branch is decreasing) [30], we assume that the initial ingestion rate is r_{init}, while the current ingestion rate is r_{cur}. Then, the computing resource that shall be reallocated to the other branches is shown in Equation 10. This adaptive strategy is very light-weight, because we can monitor the ingestion rate during batch loading and redistribute the proportion of reduced computing resources to the branch that has a higher ingestion rate. In the case of Figure 8 (b), we can keep limited hardware resource in OP2 and redistribute the rest to OP3 after processing the current batch.

$$resource_{redistribute\_i} = \frac{r_{init} - r_{curr}}{r_{init}} \cdot resource_{OPi} \tag{10}$$

Query Plan Adjustment. With reference to [30], FineStream generates not only the query plan that soon will be used in the stream processing, but also several possible alternatives. During stream processing, FineStream monitors the size of intermediate results. If the performance degrades and the size of intermediate results varies greatly, FineStream shall switch to another alternative query plan topology. In the implementation, FineStream generates three additional plans by default, and picks them based on the performance model.

6 Implementation Details

6.1 How FineStream Works

We present the system workflow in Figure 9. In Figure 9, *thread1* is used to cache input data, while *thread2* is used to process the cached data. The detailed workflow is as follows. First, when FineStream starts a new query, the dispatcher executes the query on the CPU for *batch1* and then on the GPU for *batch2*. Second, during these single-device executions, FineStream conducts online profiling, during which the operator-related data that are used to build the performance model are obtained, including the CPU and GPU performance, and bandwidth utilization. Third, with these data, FineStream builds the performance model considering branch co-running and batch pipeline. Fourth, after building the model, FineStream generates several query plans with detailed resource distribution. With the generated query plan, the dispatcher performs fine-grained scheduling for processing the following batches. When dynamic workload is detected, FineStream performs related adjustment mentioned in Section 5.3. For the operators in FineStream, we reuse the operator code from OmniDB [64]. Please note that the goal of this work is to provide a fine-grained stream processing method on integrated architectures. The same methodology can also be applied for using other OpenCL processing engine such as Voodoo [45].

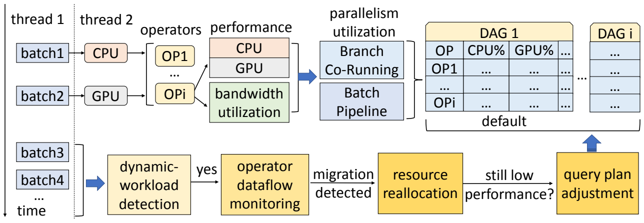


Figure 9: FineStream workflow.

Additionally, when users change the window size of a query on the fly, FineStream updates the window size parameter after the related batch processing is completed, and then continues to run with performance detection. If the performance decreases below a given value (70% of the original performance by default), FineStream re-determines the query plan with computing resource based on the parameters and the performance model.

6.2 Dispatcher

The dispatcher of FineStream is a module for assigning stream data to the related operators with hardware resources. The dispatcher has two functions. First, it splits the stream data into batches with a fixed size. Second, it sends the batches to the corresponding operators with proper hardware resources for execution. The goal of the dispatcher is to schedule operator tasks to the appropriate devices to fully utilize the capacity of the integrated architecture.

Algorithm 1 is the pseudocode of the dispatcher. When a stream is firstly presented in the engine, FineStream conducts branch co-running and batch pipeline according to the performance model mentioned in Section 5 (Lines 2 to 5). FineStream also detects dynamic workload (Lines 6 to 15). If dynamic workload is detected, FineStream conducts the related resource reallocation. If such reallocation does not help, it further conducts query plan adjustment.

Algorithm 1: Scheduling Algorithm in FineStream

```
1  Function dispatch($batch, resource, model$):
2      if $taskFirstRun$ then
3          $branchCoRun(resource, model)$
4          $batchPipeline(resource, model)$
5          $taskFirstRun = false$
       // Handling dynamic workload and query plan optimization
6      if $detectDynamicWorkload()$ then
7          $resourceReallocate()$
8          if $resourceChanged == true$ then
9              if $performanceDegrade()$ then
10                 $adjustQueryPlan()$
11                 $queryChanged = true$
12         $resourceChanged = true$
13         if $queryChanged == true$ then
14             $resourceChanged = false$
15             $queryChanged = false$
```

6.3 Online Profiling

The purpose of online profiling is to collect useful performance data to support building the performance model.

In online profiling, we have two concerns. The first is what data to generate in this module. This is decided by the performance model. These data include the data size, execution time, bandwidth utilization, and throughput for each operator on devices. The second is, to generate the data, what information we shall collect from stream processing.

FineStream performs online profiling for operators from memory bandwidth and computation perspectives.

Memory Bandwidth Perspective. Based on the above analysis, we use *bandwidth*, defined as the transmitted data size divided by the execution time, to depict the characteristics from data perspective of an operator. The transmitted data for an operator consists of input and output. The input relates to the batch while the output relates to both the operator and the batch. We define the bandwidth of the operator i on device dev in Equation 11. The parameters s_{input_i} and s_{output_i} denote the estimated input and the output sizes of the operator i, and $t_{i,dev}$ represents the execution time of the operator i on device dev.

$$bandwidth_{i,dev} = \frac{s_{input\_i} + s_{output\_i}}{t_{i,dev}} \qquad (11)$$

Computation Perspective. To depict the characteristics from the computation perspective, we use $throughput_{i,dev}$, which is defined as the total number of processed tuples n_{tuples} divided by the time $t_{i,dev}$ for operator i on device dev. All these values can be obtained from online profiling.

Table 3: The queries used in evaluation.

Query	Detail
Q1	select timestamp, category, sum(cpu) as totalCPU from TaskEvents [range 256 slide 1] group by category
Q2	select timestamp, jobID, avg(cpu) as avgCPU from TaskEvents [range 256 slide 1] where eventType == 1 group by jobId
Q3	select timestamp, eventType, userId, max(disk) as maxDisk from TaskEvents [range 256 slide 1] group by eventType, userId
Q4	select timestamp, avg (value) as globalAvgLoad from SmartGridStr [range 512 slide 1]
Q5	select timestamp, plug, household, house, avg(value) as localAvgLoad from SmartGridStr [range 512 slide 1] group by plug, household, house
Q6	(select L.timestamp, L.plug, L.household, L.house from LocalLoadStr [range 1 slide 1] as L, GlobalLoadStr [range 1 slide 1] as G where L.house == G.house and L.localAvgLoad >G.globalAvgLoad) as R – select timestamp, house, count(*) from R group by house
Q7	(select timestamp, vehicle, speed, highway, lane, direction, (position/5280) as segment from PosSpeedStr [range unbounded]) as SegSpeedStr – select distinct L.timestamp, L.vehicle, L.speed, L.highway, L.lane, L.direction, L.segment from SegSpeedStr [range 30 slide 1] as A, SegSpeedStr [partition by vehicle rows 1] as L where A.vehicle == L.vehicle
Q8	select timestamp, vehicle, count(direction) from PosSpeedStr [range 256 slide 1] group by vehicle
Q9	select timestamp, max(speed), highway, lane, direction from PosSpeedStr [range 256 slide 1] group by highway,lane,direction

7 Evaluation

7.1 Methodology

The baseline method used in our comparison is Saber [34], while our method is denoted as "FineStream". Saber is the state-of-the-art window-based stream processing engine for discrete architectures. It adopts a bulk-synchronous parallel model [66]. The whole query execution on a batch is distributed to a device, the GPU or the CPU, without further distributing operators of a query to different devices. The original CPU operators in Saber are written in Java, and we further rewrite the CPU operators in Saber in OpenCL for higher efficiency. Our comparisons to Saber examine whether our fine-grained method delivers better performance. To validate the co-running benefits of the two devices, we also measure the performance using only the CPU and the performance using only the GPU, denoted as "CPU-only" and "GPU-only". Further, to understand the advantage of using the integrated architecture to accelerate stream processing, we compare FineStream on integrated architectures with Saber on discrete CPU-GPU architectures.

Platforms. We perform experiments on four platforms, two integrated platforms and two discrete platforms. The first integrated platform uses the integrated architecture AMD A10-7850K [15], and it has 32 GB memory. The second integrated platform uses the integrated architecture Ryzen 5 2400G, which is the latest integrated architecture, and this platform has 32 GB memory. The first discrete platform is equipped with an Intel i7-8700K CPU and an NVIDIA GeForce GTX 1080Ti GPU, and along with 32 GB memory. The second discrete platform is equipped with two Intel E5-2640 v4 CPUs and an NVIDIA V100-32GB GPU, and has 264 GB memory.

Datasets. We use four datasets in the evaluation. The first dataset is Google compute cluster monitoring [2], which emulates a cluster management scenario. The second dataset is anomaly detection in smart grids [68], which is about detection in energy consumption from different devices of a smart electricity grid. The third dataset is linear road benchmark [7], which models a network of toll roads. These traces come from real-world applications, and are widely used in previous studies such as [19, 34, 39]. The fourth dataset is a synthetically generated dataset [34] for evaluating independent operators, where each tuple consists of a 64-bit timestamp and six 32-bit attributes drawn from a uniform distribution. To overfeed the system and test its performance capacity, we load the data from memory. This method avoids network being bottleneck. In practice, the system obtains stream data via network.

Benchmarks. We use nine queries to evaluate the overall performance of the fine-grained stream processing engine on the integrated architectures. Similar benchmarks have been used in [34]. The details of the nine queries are shown in Table 3. *Q1*, *Q2*, and *Q3* are conducted on the Google compute cluster monitoring dataset. *Q4*, *Q5*, and *Q6* are for the dataset of anomaly detection in smart grids. *Q7*, *Q8*, and *Q9* are for the dataset from the linear road benchmark.

Dynamic Workload Generation. We use the datasets and benchmarks to generate dynamic workload. For the first dataset of cluster monitoring, the seventh attribute of *category* gradually changes from type 1 to type 2 within 10,000 batches. We use the query *Q1* for illustration, and we denote it as *T1*. Similar evaluations are also conducted on the second dataset of smart grid with the query *Q5*, which is denoted as *T2*, and the third dataset of linear road benchmark with the query *Q8*, which is denoted as *T3*.

7.2 Performance Comparison

Throughput. We explore the throughput of FineStream for the nine queries. Figure 10 shows the processing throughput of the best single device, Saber, and FineStream for these queries on both the A10-7850K and Ryzen 5 2400G platforms. Please note that the y-axis of the figure is in log scale. We have the following observations. First, on the A10-7850K platform, FineStream achieves 88% throughput improvement over the best single device performance on average; compared to Saber, FineStream achieves 52% throughput improvement. Because of the efficient CPU and GPU co-running, FineStream nearly doubles the performance compared to the method of using only a single device. Because FineStream adopts the continuous operator model where each operator could be scheduled on its preferred device, FineStream utilizes the integrated architecture better than Saber that uses the bulk-synchronous parallel model. Such result clearly shows the advantage of fine-grained stream processing on the integrated architecture. Second, on the Ryzen 5 2400G platform, all hardware configurations have been upgraded in comparison with A10-7850K, especially the CPUs; the CPU-only throughput on Ryzen 5 2400G is much higher than that on A10-7850K. Moreover, Saber achieves a 56% throughput improvement compared to the throughput of the best single device, and FineStream is still 14% higher than Saber on this platform. Similar phenomena have also been observed in [58–60].

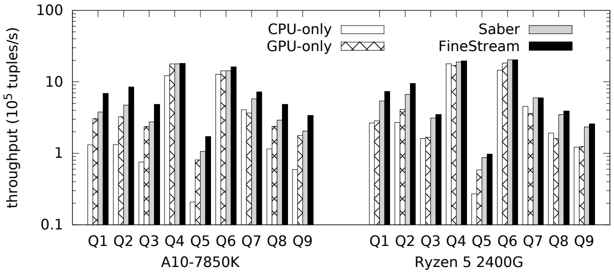


Figure 10: Throughput of different queries.

Latency. Figure 11 reports the latency of different queries on the integrated architectures. In this work, latency is defined as the end-to-end time from the time a query starts to the time it ends. FineStream has the lowest latency among these methods. First, on the A10-7850K platform, FineStream's latency is 10% lower than that of the best single device, and 36% lower than the latency of Saber. Second, on Ryzen 5 2400G platform, it is 2% lower than that of the best single device, and 9% lower than that of Saber. The reason is that FineStream considers device preference for operators and assigns the operators to their suitable devices. In this way, each batch can be processed in a more efficient manner, leading to lower latency.

Profiling. We show the relationship between throughput and latency of both FineStream and Saber in Figure 12. Figure 12 shows that queries with high throughput usually have low latency, and vice versa.

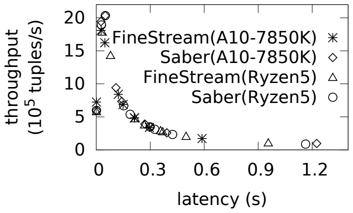


Figure 12: Throughput vs. latency.

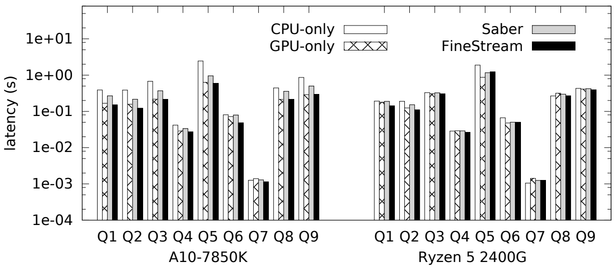


Figure 11: Latency of different queries.

We further study the CPU and GPU utilization of Saber and FineStream, and use the A10-7850K platform for illustration, as shown in Figure 13. In most cases, FineStream utilizes the GPU device better on the integrated architecture. As for *Q*4, the CPU processes most of the workload. On average, FineStream improves 23% GPU utilization compared to Saber, and have roughly the same CPU utilization as Saber. Since FineStream achieves better throughput and latency than Saber, such utilization results indicate that FineStream generates effective strategies in determining device preferences for individual operators.

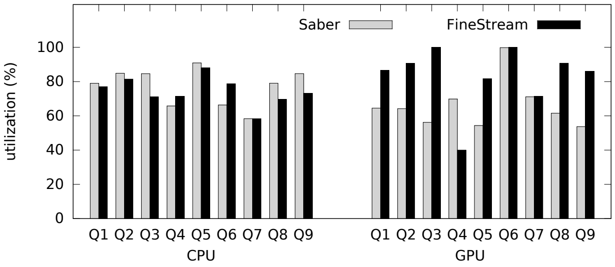


Figure 13: Utilization of the integrated architecture.

7.3 Comparison with Discrete Architectures

In this part, we compare FineStream on the integrated architectures and Saber on the discrete architectures from three perspectives: performance, price, and energy-efficiency.

Performance Comparison. The current GPU on the integrated architecture is less powerful than the discrete GPU, as mentioned in Section 2.1. The discrete GPUs exhibit 1.8x to 5.7x higher throughput than the integrated

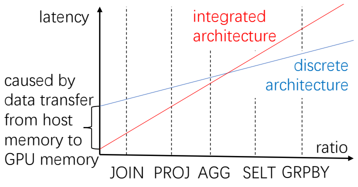


Figure 14: Latency comparison of different operators.

architectures, due to the more computational power of discrete GPUs. However, the integrated architecture demonstrates lower processing latency compared to the discrete architecture when the data transmission cost between the host memory and GPU memory in the workload is significant. For example, the latencies for *projection*, *selection*, *aggregation*, *group-by*, and *join* are 0.6, 1.5, 1.0, 10.6, and 1924.5 ms on Ryzen 5 2400G platform, while 1.1, 1.2, 1.2, 1.6, and 7600.1 ms on GTX 1080Ti platform; these operators are distributed in Figure 14, where *join* (JOIN), *projection* (PROJ), and *aggregation* (AGG) achieve lower latency on the integrated architecture, while *selection* (SELT), and *group-by* (GRPBY) prefer the discrete architecture. The x-axis represents the ratio of $m_{compute}/(s_{write}+s_{read})$ where $m_{compute}$ denotes the kernel computation workload size, and t_{write} and s_{read} denote the data transmission sizes from the host memory to the GPU memory and from the GPU memory to the host memory via PCI-e. For further explanation, to execute a kernel on discrete GPUs, the execution time t_{total} includes 1) the time t_{write} of data transmission from the host memory to the GPU memory via PCI-e, 2) the time $t_{compute}$ for data processing kernel execution, and 3) the time t_{read} of data transmission from the GPU memory to the host memory. As for executing a kernel on the integrated architecture, although its $t_{compute}$ is longer than that on discrete GPUs, its t_{write} and t_{read} can be avoided. For the queries in Table 3, the data movement overhead on discrete architectures ranges from 31 to 62%.

Price-Throughput Ratio Comparison. FineStream on integrated architectures shows a high price-throughput ratio, compared to Saber on the discrete architectures. The price of the 1080Ti discrete architecture is about 7x higher than that of the A10-7850K integrated architecture, and the price of the V100 discrete architecture is about 64x higher than that of the Ryzen 5 2400G integrated architecture. Figure 15 shows the comparison of their price-throughput ratio. On average, FineStream on the integrated architectures outperforms Saber on the discrete architectures by 10.4x.

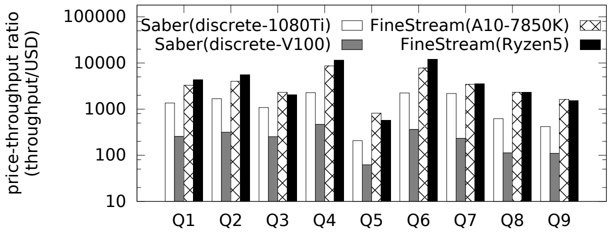


Figure 15: Comparison of price-throughput ratio.

Energy Efficiency Comparison. We also analyze the energy efficiency of FineStream and Saber. The Thermal Design Power (TDP) is 95W on A10-7850K, and 65W on Ryzen 5 2400G. For the 1080Ti platform, the TDP of the Intel i7-8700K CPU and NVIDIA GTX 1080Ti GPU are 95W and 250W, respectively. For the V100 platform, the TDP of the Intel E5-2640 v4 CPU and NVIDIA V100 GPU are 90W and 300W, respectively. Similar to [61], we use performance per Watt to define energy efficiency. On average, FineStream on the integrated architectures is 1.8x energy-efficient than Saber on the discrete architectures.

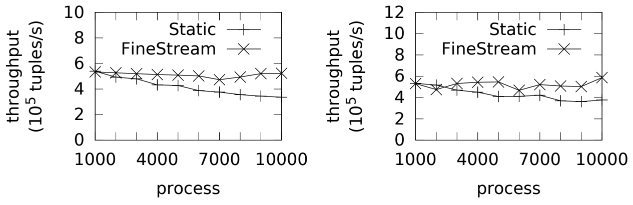
7.4 Handling Dynamic Workload

In this section, we discuss how to handle dynamic workloads. To demonstrate the capability of FineStream to handle dynamic workload, we evaluate FineStream on the dynamic workloads mentioned in Section 7.1. On average, FineStream achieves a performance of 323,727 tuples per second, which outperforms the static method (we denote "static" for FineStream without adapting to dynamic workload) by 28.6%, as shown in Table 4.

Table 4: Throughput of the queries on dynamic workloads.

Dynamic	A10-7850K (10^5 tuples/s)		Ryzen 5 2400G (10^5 tuples/s)	
Workload	Static	FineStream	Static	FineStream
T1	4.2	5.1	4.4	5.1
T2	0.8	1.2	1.1	1.5
T3	1.9	2.8	2.7	3.7

We use *T1* as an example, and show the detailed throughput along with the number of batches in Figure 16. In the timeline process, the static method decreases due to the improper hardware resource distribution. As for FineStream, the hardware computing resources can be dynamically adjusted according to the data distribution, so the performance does not decline with the changes.



(a) A10-7850K.

(b) Ryzen 5 2400G.

Figure 16: Throughput of *T1* on dynamic workloads.

7.5 Detailed Analysis

Performance Model Accuracy. In stream processing, after each batch is processed, we use the measured batch processing speed to correct our model. We use the example of *Q1* for illustration, as shown in Figure 17. We use the percent deviation to measure the accuracy of our performance model. The percent deviation is defined as the absolute value of the real throughput minus the estimated throughput, divided by the real throughput. The smaller the percent deviation is, the more accurate the predicted result is. The deviation decreases as the number of processed batches increases. After 20 batches are processed, we can reduce the deviation to less than 10%. Please note that in stream processing scenarios, input tuples are continuously coming, so the time for correcting performance prediction can be ignored in stream processing. For dynamic workload, the accuracy also depends on the intensity of workload changes.

Runtime Overhead Analysis. FineStream incurs runtime overhead in the batch processing phase from two aspects. First, it detects whether the input stream belongs to dynamic workload, which causes time overhead. Second, the scheduling also takes time. In our evaluation, we observe that the time overhead accounts for less than 2% of the processing time, which can be ignored in stream processing.

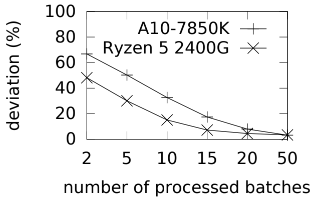


Figure 17: Deviation of *Q1*.

8 Related Work

Parallel stream processing [4, 10, 12, 21, 28, 29, 34, 43, 46, 63], query processing [9, 14, 16, 20, 22, 56], and heterogeneous systems [11, 17, 24, 26, 32, 35–38, 45, 48, 50] are hot research topics in recent years. Different from these works, FineStream targets sliding window-based stream processing, which focuses on window handling with SQL and dynamic adjustment. GPUs have massive threads and high bandwidth, and have emerged to be one of the most promising heterogeneous accelerators to speedup stream processing. Verner et al. [54] presented a stream processing algorithm considering various latency and throughput requirements on GPUs. Alghabi et al. [5] developed a framework for stateful stream data processing on multiple GPUs. De Matteis et al. [25] developed Gasser system for offloading operators on GPUs. Pinnecke et al. [44] studied how to efficiently process large windows on GPUs. Chen et al. [23] extended the popular stream processing system, Storm [1], to GPU platforms. Augonnet et al. [8] explored data-aware task scheduling for multi-devices, which can be integrated into this work. FineStream differs from those previous works in two aspects: firstly on integrated architectures, and secondly for SQL streaming processing.

The closest work to FineStream is Saber [34], which aims to utilize discrete CPU-GPU architectures. Saber [34] adopts a *bulk-synchronous parallel* model [53, 66], where the whole query (with multiple operators) on each batch of input data is dispatched on one device. Such a mechanism naturally minimizes the communication overhead among operators inside the same query. It is hence suitable in discrete CPU-GPU architectures, where PCI-e overhead is significant and shall be avoided as much as possible. However, it may result in suboptimality in integrated architectures for mainly two reasons. First, it overlooks the performance difference between different devices for each operator. Second, the communication overhead between the CPU and the GPU in integrated architectures is negligible. Targeting at integrated architectures, FineStream adopts continuous operator model [53, 66], where each operator of a query can be independently placed at a device. We further build a performance model to guide operator-device placement optimization. It is noteworthy that our fine-grained operator placement is different from classical placement strategies for general stream processing [18, 19, 40, 51, 65] for their different design goals. In particular, most prior works aim at reducing communication overhead among operators, which is not an issue in FineStream. Instead, FineStream needs to take device preference into consideration during placement optimization, which has not been considered before.

9 Conclusion

Stream processing has shown significant performance benefits on GPUs. However, the data transmission via PCI-e hinders its further performance improvement. This paper revisits window-based stream processing on the promising CPU-GPU integrated architectures, and with CPUs and GPUs integrated on the same chip, the data transmission overhead is eliminated. Furthermore, such integration opens up new opportunities for fine-grained cooperation between different devices, and we develop a framework called FineStream for fine-grained stream processing on the integrated architecture. This study shows that integrated CPU-GPU architectures can be more desirable alternative architectures for low-latency and high-throughput data strream processing, in comparison with discrete architectures. Experiments show that FineStream can improve the performance by 52% over the state-of-the-art method on the integrated architecture. Compared to the stream processing engine on the discrete architecture, FineStream on the integrated architecture achieves 10.4x price-throughput ratio, 1.8x energy efficiency, and can enjoy lower latency benefits.

Acknowledgments

We sincerely thank our shepherd Sergey Blagodurov and the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (No. 61732014, 61802412, 61972402, and 61972403), and Beijing Natural Science Foundation (No. L192027), and is also supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Xiaoyong Du is the corresponding author of this paper.

References

[1] Apache Storm. http://storm.apache.org/.

[2] More google cluster data. https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[3] The Compute Architecture of Intel Processor Graphics Gen7.5. https://software.intel.com/.

[4] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *SOSP*, 2017.

[5] Farhoosh Alghabi, Ulrich Schipper, and Andreas Kolb. A scalable software framework for stateful stream data processing on multiple gpus and applications. In *GPU Computing and Applications*. 2015.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 2006.

[7] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *PVLDB*, 2004.

[8] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *ICPADS*, 2010.

[9] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[10] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *INFOCOM*, 2016.

[11] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In *USENIX ATC*, 2017.

[12] Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilowska, Karsten Schwan, and Adam Allred. Appflux: Taming app delivery via streaming. *Proc. of the Usenix TRIOS*, 2015.

[13] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's First 64-bit ARM Processor. *Micro*, 2015.

[14] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 2008.

[15] Dan Bouvier and Ben Sander. Applying AMD's Kaveri APU for heterogeneous computing. In *Hot Chips Symposium*, 2014.

[16] Sebastian Breß and Gunter Saake. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *PVLDB*, 2013.

[17] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. Mobirnn: Efficient recurrent neural network execution on mobile GPU. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, 2017.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[19] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[20] Badrish Chandramouli, Raul Castro Fernandez, Jonathan Goldstein, Ahmed Eldawy, and Abdul Quamar. Quill: efficient, transferable, and rich analytics at scale. *PVLDB*, 2016.

[21] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. Accurate latency estimation in a distributed event processing system. In *ICDE*, 2011.

[22] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 2014.

[23] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *Big Data*, 2015.

[24] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment*, 2019.

[25] Tiziano De Matteis, Gabriele Mencagli, Daniele De Sensi, Massimo Torquati, and Marco Danelutto. GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs. *IEEE Access*, 2019.

[26] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *HPDC*, 2019.

[27] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *Micro*, 2017.

[28] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. NVStream: accelerating HPC workflows with NVRAM-based transport for streaming objects. In *HPDC*, 2018.

[29] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *USENIX ATC*, 2019.

[30] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *TPDS*, 2013.

[31] Younghwan Go, Muhammad Asim Jamshed, Young-Gyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI*, 2017.

[32] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*, 2020.

[33] Chandima HewaNadungodage, Yuni Xia, and John Jaehwan Lee. GStreamMiner: a GPU-accelerated data stream mining framework. In *CIKM*, 2016.

[34] Alexandros Koliousis and et al. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, 2016.

[35] Xinyu Li, Lei Liu, Shengjie Yang, Lu Peng, and Jiefan Qiu. Thinking about A New Mechanism for Huge Page Management. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019.

[36] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *TPDS*, 2019.

[37] Alexander M Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. In *VTDC*, 2011.

[38] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *HPCA*, 2015.

[39] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing*, 2014.

[40] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *ICDM Workshops*, 2010.

[41] Dong Nguyen and Jongeun Lee. Communication-aware mapping of stream graphs for multi-GPU platforms. In *CGO*, 2016.

[42] John Nickolls and William J Dally. The GPU computing era. *Micro*, 2010.

[43] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[44] Marcus Pinnecke, David Broneske, and Gunter Saake. Toward GPU Accelerated Data Stream Processing. In *GvD*, 2015.

[45] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 2016.

[46] Arosha Rodrigo, Miyuru Dayarathna, and Sanath Jayasena. Latency-Aware Secure Elastic Stream Processing with Homomorphic Encryption. *Data Science and Engineering*, 2019.

[47] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. Achieving exascale capabilities through heterogeneous computing. *Micro*, 2015.

[48] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *TOCS*, 2016.

[49] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[50] Zhi Tang and Youjip Won. Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture. In *2011 First International Conference on Data Compression, Communications and Processing*, 2011.

[51] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *SIGMOD*, 2014.

[52] Sid Touati and Benoit Dupont De Dinechin. *Advanced Backend Code Optimization*. John Wiley & Sons, 2014.

[53] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[54] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *ICS*, 2011.

[55] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Bradford M Beckmann, William C Brantley, Joseph L Greathouse, Wei Huang, Arun Karunanithi, et al. Design and Analysis of an APU for Exascale Computing. In *HPCA*, 2017.

[56] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *HPCA*, 2009.

[57] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[58] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, and Wenguang Chen. FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures. In *CGO*, 2017.

[59] Feng Zhang, Bo Wu, Jidong Zhai, Bingsheng He, Wenguang Chen, and Xiaoyong Du. Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. *TKDE*, 2019.

[60] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. Understanding co-running behaviors on integrated cpu/gpu architectures. *TPDS*, 2017.

[61] Kai Zhang, Jiayu Hu, Bingsheng He, and Bei Hua. DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *ICDE*, 2017.

[62] Kai Zhang, Jiayu Hu, and Bei Hua. A holistic approach to build real-time stream processing system with GPU. *JPDC*, 2015.

[63] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *ICDE*, 2017.

[64] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 2013.

[65] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. Briskstream: Scaling Data Stream Processing on Multicore Architectures. In *SIGMOD*, 2019.

[66] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. Hardware-conscious stream processing: A survey. *SIGMOD Rec.*, 2020.

[67] Yongpeng Zhang and Frank Mueller. GStream: A general-purpose data streaming framework on GPU clusters. In *ICPP*, 2011.

[68] Holger Ziekow and Zbigniew Jerzak. The DEBS 2014 grand challenge. In *DEBS*, 2014.

OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices

Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

Abstract

NVMe is widely used by diverse types of storage and non-volatile memories subsystems as a de-facto fast I/O communication interface. Industries secure their own intellectual property (IP) for high-speed NVMe controllers and explore challenges of software stack with future fast NVMe storage cards. Unfortunately, such NVMe controller IPs are often inaccessible to academia. The research community, however, requires an open-source hardware framework to build new storage stack and controllers for the fast NVMe devices.

In this work, we present OpenExpress, a fully hardware automated framework that has no software intervention to process concurrent NVMe requests while supporting scalable data submission, rich outstanding I/O command queues, and submission/completion queue management. OpenExpress is available to download and offers a maximum bandwidth of around 7GB/s without a silicon fabrication.

1 Introduction

NVM Express (NVMe) is successful in bringing diverse solid state drive (SSD) and non-volatile memory (NVM) technologies closer to CPUs. NVMe supports many current I/O submissions and completions for higher throughput compared to traditional block storage interfaces. The storage-level firmware and host-side software can synchronize their communication over a flexible queueing method [26]. NVMe also provides a data transfer mechanism that allows the underlying NVMe devices to leverage abundant system memory resources of the host via physical region page (PRP) lists.

As NVMe devices have been widely used in a broad spectrum of computing domains [10, 17, 21], storage vendors secure their own *intellectual property* (IP) cores to build high-speed NVMe controllers. The NVMe specification stipulates these NVMe controller IP cores to be implemented in either firmware or hardware [26]. Considering the design flexibility of the underlying SSD architecture, many storage vendors and communities typically implement the controller IP cores as firmware [20,31,39]. Even though firmware-based controllers are sufficient to manage flash, their firmware execution can be a critical performance bottleneck when used in conjunction with faster NVM media, such as phase change memory (PRAM) and magnetoresistive memory (MRAM) [8, 23]. Since the latency of the emerging fast NVMs is shorter than that of flash by an order of magnitude, CPU cycles to execute the firmware cannot be hidden behind or overlapped with I/O burst times of the underlying NVM backend.

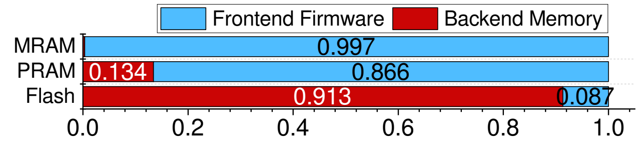
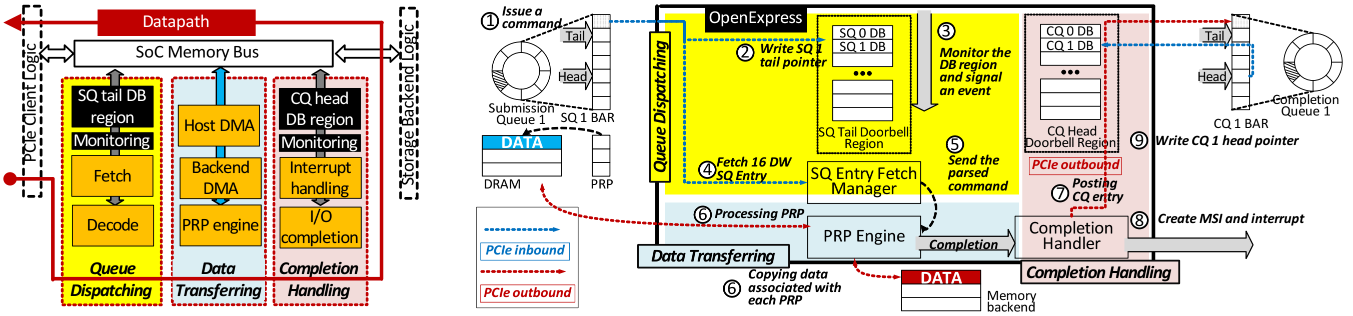


Figure 1: Decomposition comparison for execution cycles with different non-volatile memories.

To illustrate this overhead, we perform an emulation study to decompose the latency of an NVMe SSD into two parts: i) NVMe firmware execution time (*Frontend*) and ii) NVM backend access latency (*Backend*). In this study, we intermix read and write requests (1:1) and issue them to our PCIe hardware platform; this platform executes the NVMe firmware on a 1 GHz processor and emulates three different types of NVM backends, each employing 16 flash chips [14], and 16 PRAM [23] chips, and 16 MRAM chips [8], respectively. As shown in Figure 1, the access latency of flash-based backend contributes 91% of the total NVMe service time, which makes it possible to hide the front-end firmware latency by interleaving with flash I/O bursts. In contrast, the firmware execution cannot be overlapped with the I/O burst times of PRAMs and MRAMs as its execution time accounts for 86% and 99% of the total I/O latency. To address this challenge, industries have begun to employ hardware automated NVMe controller IP cores [9,24]. For example, recent state-of-the-art prototypes [9] automate NVMe control logic in an SSD to achieve 1 million I/O per second (IOPS).

Unfortunately, such NVMe hardware IP cores are prohibited to access and redesign from academia. There exist a few third-party vendors that do hardware IP business [7, 15, 16]. However, their IP cores require expensive “per-month” unitary prices, ranging from \$30K $\sim$ \$45K; the price of single-use source code is around \$100K. There are even restrictions on modifying the hardware IP and architecture, which is inadmissible in research communities. Because of such unapproachability and high costs of NVMe IP cores, many recent studies that redesign the existing storage stack, such as bypass kernel designs or NVM file systems implemented atop a fast NVMe device, are mostly performed by a simulation [11, 12, 19, 34] or an emulation [22, 37, 40]. In addition, most academic studies related to SSDs such as firmware algorithm developments [4, 6, 25, 29] and cross-layer optimizations [5, 18, 38], are all performed by the simulation/emulation.



a) OpenExpress overview.

b) Functional descriptions for OpenExpress.

Figure 2: Overview and hardware details of the proposed OpenExpress.

While there are benefits for the simulation/emulation-based studies (e.g., fast prototyping and validation), many researchers still require real NVMe devices that they can modify to develop full-system prototypes and perform end-to-end performance evaluation with no strings attached. In this work, we propose *OpenExpress*, a framework that fully automates NVMe control logic in hardware to make it possible to build customizable NVMe devices. To the best of our knowledge, OpenExpress is only the open research framework that realizes NVMe hardware accelerator IPs [1], which does not require any software intervention to process concurrent read and write NVMe requests. Considering diverse demands of the academic research, OpenExpress leverages multiple DRAM channels and modules as storage backend and supports scalable data submission, rich outstanding NVMe commands and submission/completion queue management.

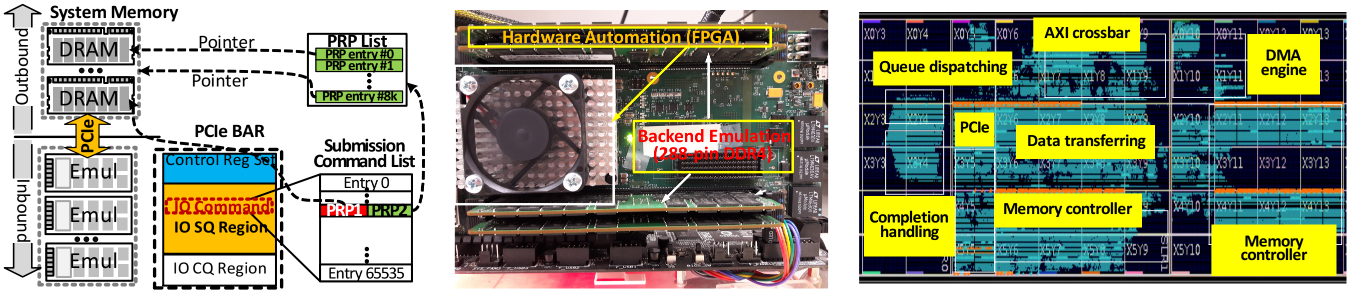
We prototype OpenExpress with a commercially available Xilinx FPGA board [3] and optimize all its logic components to operate at a high frequency. Even though it is a well-known fact that FPGAs are slower than CPUs, we demonstrate that OpenExpress can expose the true performance of backend memory modules to users over PCIe. At its peak, OpenExpress provides an access bandwidth of 7GB/s. Our evaluation also shows that OpenExpress can, on average, exhibit 76.3% better performance than an Optane SSD [13], which is one of the fast NVMe devices in the market.

2 Module Design of OpenExpress

Overview. Figure 2a illustrates a high-level view of OpenExpress, which is composed of three groups of hardware modules: i) *queue dispatching*, ii) *data transferring*, and iii) *completion handling*. At the beginning of OpenExpress's datapath, queue dispatching hardware modules fetch and decode all incoming I/O requests by monitoring multiple NVMe *submission queues* (SQs). Data transferring hardware modules then check the host system memory over *physical region page* (PRP) lists, and perform DMA for both host and backend memories. Once the data is successfully served by the backend, OpenExpress's completion handling modules manage interrupts and NVMe *completion queues* (CQs) over a set of registers (*doorbells*) mapped to a system-on-chip (SoC) internal memory bus. The details of the three different hardware module groups are explained in Figure 2b.

Queue dispatching modules. [① ∼ ⑤] of Figure 2b describe the queue dispatching functions that OpenExpress implements over hardware, including a set of procedures for automatic NVMe doorbell monitoring, command fetches, and command decodes. Specifically, when the host issues an NVMe command by pushing an SQ entry into the per-core queue [①], it should also write (i.e., ring) the corresponding entry of the doorbell (DB). Thus, OpenExpress creates a DB entry per queue by grouping two sets of DBs, each keeping track of a *tail* pointer per SQ and a *head* pointer per CQ. We map these sets of DBs, called *DB region*, to the internal SoC memory bus address space, and in parallel, we expose it to a PCIe's *base address register* (BAR) [②]. Since our fetch and decode modules can detect any DB updates (made by the host) through the memory-mapped DB region, OpenExpress is able to synchronize the host NVMe queue states with its internal NVMe queue states. Note that ringing a DB at the host side is the same as writing a PCIe packet (i.e., PCIe inbound) in NVMe. Therefore, OpenExpress can simply catch each ringing event by monitoring the memory bus address space mapped to the DB region for changes. Whenever OpenExpress observes an update of a DB that exists in the SQ-tail DB region and CQ-head DB region, it generates an event signal [③]. This event signal includes information relevant to the target SQ, and is passed to the underlying SQ entry *fetch manager*. Since multiple event signals can arise simultaneously, our fetch manager arbitrates between different NVMe queues in a round robin manner as specified by NVMe 1.4 [26]. The fetch manager then checks the delivered information and copies the 16-double-word (DW) NVMe command, associated with the target SQ entry, from the host-side system memory. After fetching the command, OpenExpress can see all relevant request information, such as the operation code (opcode), logical block address (LBA), block size, and PRPs [④]. Our fetch manager finally parses

[1] The intellectual property (IP) cores and/or register transfer level (RTL) code our OpenExpress is available to download for non-commercial and academic uses from http://openexpress.camelab.org.



a) Data transferring (PRP).

b) Reference board for our prototype.

c) Floorplan of OpenExpress.

Figure 3: Implementation View of OpenExpress.

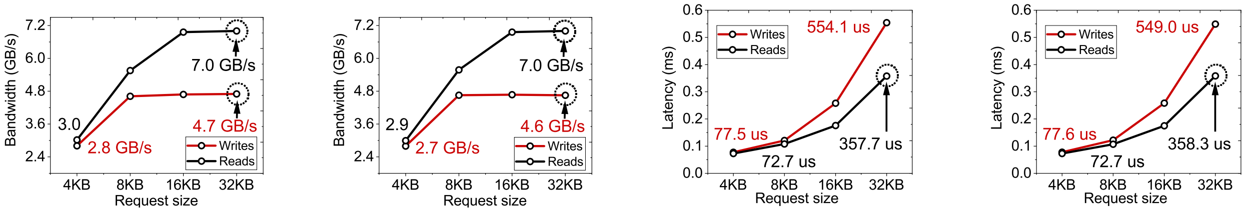
the request information and forwards it to the underlying PRP engine. To enhance responsiveness of the storage frontend (i.e., queue dispatching), the fetch manager pipelines its processing of other remaining requests with other operations of data transferring and completion handling modules [⑤].

Data transferring modules. To handle page-aligned data, our PRP engine accesses the location of the host-side system memory via a PRP, which in turn generates an out-bound PCIe packet. The PRP engine then initiates data transfer by setting the source address (i.e., host-side system memory) and the destination address (i.e., OpenExpress's memory) of our backend DMA engine [⑥]. The PRP engine copies the data from the host DRAM to our DMA engine for each 512B chunk to make device memory compatible with sector-based block I/O services. On the other hand, the DMA engine employs multiple DMA IP cores (as many as FPGA's backend memory channels) to fully parallelize data transfers. Specifically, the DMA engine splits the data of a single request into multiple small-sized data chunks and spreads them across the multiple DMA engine IP cores of the target backend. Doing so will help shorten the I/O latency of underlying NVM emulation memories. Note that the payload size of a PCIe packet is 4KB. Thus, if the request offset is not aligned to a 4KB boundary, the target data can exist in two different system memory pages of the host-side DRAM. To manage this case, as shown in Figure 3a, the PRP engine fetches the target data from two different host locations, referred by *PRP1* and *PRP2*, respectively. Specifically, if the request size is greater than a page, the PRP engine brings a page and the PRP list, indicated by PRP1 and PRP2, respectively, from the host DRAM and parses each entry of the list. The PRP engine then traverses all entries of the PRP list and transfers the data of each entry from the host system memory pages to the underlying memories. Once the data transfers are completed by the DMA engine, the PRP engine signals the completion handler, which in turn creates a completion request (CQ entry), and manages an *message-signaled interrupt* (MSI) packet corresponding to the submission request.

Completion handling modules. The functions of completion handling that we implement include I/O completion (CQ management) and interrupt processing. Based on the NVMe specification states, an NVMe queue is logically composed by a pair of an SQ and a CQ. Thus, to make the NVMe queues easy to manage, one may locate the SQ-tail DB region with the CQ-head DB region, side-by-side. However, as in-flow velocities of NVMe submissions and NVMe completions are asymmetric, a physical integration of the SQ-tail DB region and CQ-head DB region close together is not ideal from both performance and implementation angles. Thus, OpenExpress completely separates the SQ-tail DB region and CQ-head DB region. To pair OpenExpress's CQ and SQ status in an appropriate manner, we introduce a completion handler that automatically detects which CQ is engaged with the I/O request finished to process by the data transferring modules. Specifically, the detection is performed by checking the corresponding request information, which is forwarded from the SQ entry fetch manager of the queue dispatching modules. Once the completion handler finds out the appropriate CQ, the handler posts the target CQ entry by directly writing it to the corresponding host-side system memory (BAR) [⑦]. The completion handler then interrupts the host by pushing (writing) a PCIe packet to an MSI region that the host driver manages to inform the host of the I/O service completion [⑧]. The host driver later invokes its own interrupt service routine, which will notify the completion to the user who requested such I/O service. The host finally terminates the I/O service and synchronizes its CQ state with OpenExpress by updating the CQ-head DB region [⑨]. Note that queue synchronization means that the completion handler releases the target SQ and CQ entry pair from the corresponding NVMe queue. In this way, the internal CQ and SQ states of our OpenExpress can be consistent with the host-side SQ and CQ states of NVMe driver(s).

3 Hardware Prototyping

We implement OpenExpress on an Xilinx FPGA board [3] that employs an UltraScale [36] chip and PCIe Gen3 interface, which are shown in Figure 3b. We use four 288-pin DDR4 dual in-line memory modules (DIMMs) for storage-backend emulation. In our implementation, the FPGA logic modules are classified into two: frontend and backend automa-



(a) Sequential bandwidth.

(b) Random bandwidth.

(c) Sequential latency.

(d) Random latency.

Figure 4: Bandwidth and latency performance for multi-block I/Os.

tions. The frontend automation contains most logic modules for queue dispatching and completion handling. It also exposes PCIe BAR registers and maps them into the internal advanced extensible interface (AXI) memory crossbar [1,35]. The queue dispatching IP cores fetch and decode NVMe commands issued by the host while completion handling logic manages all aspects of I/O request completion, including interrupt handling. The completion handling logic also maintains all the contexts to automatically pair different SQs and CQs by collaborating with queue dispatching cores. On the other hand, the backend automation consists of data transferring IP cores, DDR4 memory controllers, and a DMA engine. The data transferring IP cores traverse all PRP entries and migrate the corresponding data between the host's system memory and FPGA memory modules through the DDR4 memory controllers and DMA engine.

Frequency tuning. To satisfy diverse demands of NVMe-related academic research, employing reconfigurable hardware such as FPGA is essential. However, its slow clock speed is one of challenges for our prototyping to overcome. We therefore minimize the number of connections among all logic modules to make OpenExpress operate with the highest frequency of our FPGA platform. Specifically, in our implementation, the logic modules are directly engaged one-to-one, and their input and output ports are connected in a undirectional manner. The target logic can directly retrieve any necessary information that is not forwarded from a source module by accessing the memory address space of the AXI crossbar. Note that all these hardware IP cores process different parts of I/O request(s) in a pipelined manner to improve I/O bandwidth.

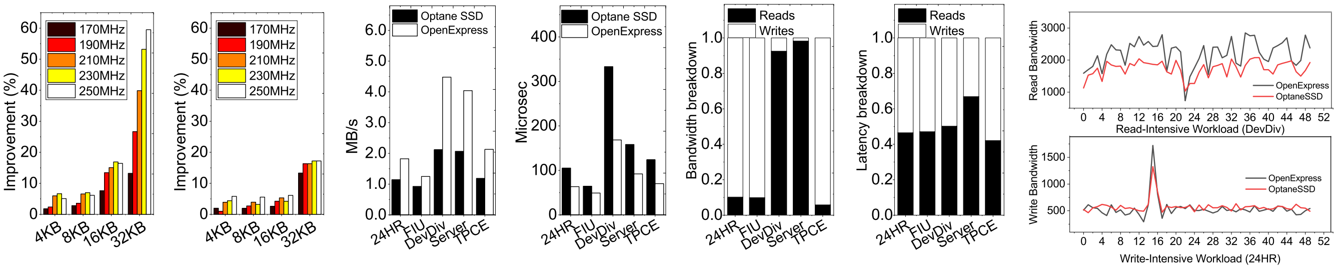
Our frequency tuning process involves gradual trial-and-error attempts to reduce the amount of route delay in a critical path. Since the delay is not straightly related to logic design, it cannot be reduced by revamping only the design itself. Instead, we perform multiple iterations of the FPGA implementation process (i.e., physical design layout decision), including a translate, map, and place-and-route (PAR) for different high-clock frequencies, ranging from 50MHz to 250 MHz. Typically, applying a higher clock frequency introduces different timing constraint violations. Unfortunately, fixing the timing violation of one core has an impact on other cores' timing characteristics. Thus, we separate the memory controller IPs from OpenExpress logic modules to the extent permitted by each each module's timing constraint, and then, we group the logic together by considering a boundary of super logic region (SLR) that contains multiple fabric modules and IP cores. The result of our design is a physical layout that is able to successfully operate at the highest frequency of the target FPGA platform as shown in Figure 3c. While all the logic modules of OpenExpress are located around PCIe and AXI crossbar, the modules associated with data transferring and DMA engine, including the memory controller IP cores, occupy only two SLRs. Lastly, the memory backend is emulated as NVMe's low-level storage complex over four channel FPGA memory IP cores. Note that, while the current prototype uses DRAM memory channels and components for the storage backend, OpenExpress IP cores can be integrated with other persistent memories.

4 Evaluation

The host system that we tested employs 8-core, 3.3GHz Intel Skylake-X processor (i9-9820X) with 32GB DDR4. Our OpenExpress prototype is attached to the host over PCIe Gen3 [30]. In this evaluation, we mainly focus on demonstrating that OpenExpress can be adopted in a real system as an open-source hardware research platform.We characterize the performance behaviors of OpenExpress (Section 4.1) and then show the performance enhancement by tuning the FPGA clock frequency (Section 4.2). We also compare the performance of OpenExpress with that of an Optane SSD [13] under diverse real workloads [32]. Note that we do not claim that OpenExpress can be faster than other fast NVMe devices. Instead, the evaluation results show that, despite the slow clock-frequency, an FPGA-based design and implementation for NVMe IP cores can offer good performance to make it a viable candidate for use in storage systems research. We use FIO [2] and execute diverse types of real workloads [32] for the user-level. We observe that the computation of a single I/O thread is insufficient to extract the maximum performance of OpenExpress. Thus, we execute FIO with a ten I/O threads, each having its own NVMe queue and I/O workload execution.

4.1 OpenExpress Characterization

Bandwidth. Figures 4a and 4b show the bandwidth trends of sequential and random I/O requests whose size varies ranging from 4 KB to 32KB, respectively. We observe that all device tests exhibit the best performance at the queue depth 8, and thus, we present only the results at such queue depth. There



(a) Read imprv. (b) Write imprv. (c) Bandwidth. (d) Latency. (e) BW. brkdown. (f) Lat. brkdown. (g) Random bandwidth.

Figure 5: Performance improvement analysis and in-depth performance comparison analysis with real workloads

is no much bandwidth difference between reads and writes at the page-size (4KB) I/O service; 3.0 GB/s and 2.8 GB/s for sequential reads and writes, respectively. As the I/O request size increases, the read and write performance improve. Note that OpenExpress reaches the maximum bandwidth with 16KB-sized I/O services; the write and read maximum bandwidth for both random and sequential are 4.6∼4.7GB/s and 7.0 GB/s, respectively. Interestingly, as the request size increases, the read bandwidth gets better than the write bandwidth. This is because of PCIe packet management issues, impacted by what we tuned for a high FPGA clock frequency, which will be analyzed in details in Section 4.2. Generally speaking, the data and command movement of reads are not congested as much as that of writes; all doorbell register updates, and command/write data payloads come together through PCIe inbound links.

Latency. Figures 4c and 4d present the latency of sequential and random I/O requests, respectively. Like the bandwidth behaviors of OpenExpress, the trend of latency curve with varying request sizes for sequential and random is similar to each other; for both reads and writes, their latency increases as the request size grows; the average latency of page-sized reads and writes is 72.7 us and 77.5 us, respectively. This is because OpenExpress exhibits the same execution latency for data transferring at a same queue depth condition. In addition, the latency of backend memory accesses with a bulk of pages does not depend on their access patterns. Even though write latency characteristics are similar to those of reads for 4∼8 KB sized requests, as increasing the I/O sizes, the latency disparity becomes more distinguishable. Specifically, the read and write latency of 32KB-sized requests is 358 us and 551.5 us, on average, respectively; for both sequential and random patterns, the writes are 54% slower than reads, on average.

Note that the latency of page-sized requests is observed with high queue depths to achieve the best bandwidth. While bandwidth is the matter on block storage, OpenExpress can reduce the latency by 62% with single queue depth operations (by sacrificing the bandwidth). Specifically, the latency of 4KB-sized requests (72.7us ∼ 77.5us) are evaluated with 8 queue depths to achieve the max bandwidth that OpenExpress can offer. Optane SSD's latency for 4KB-sized requests is 120 ∼ 150us with those queue depths. The user-level latency of 4KB-sized requests for OpenExpress is 27us ∼ 30us with a single queue-depth, which is similar to the latency that Optane SSD provides.

4.2 Frequency Tuning and Real Workload

FPGA clock frequency optimization. Figure 5a and 5b show the percentage of bandwidth improvement (as representative) by tuning the FPGA clock frequency, ranging from 150 MHz to 250 MHz. We normalize all the results to the bandwidth that we can achieve by executing OpenExpress at 150 MHz. As shown in Figure 5b, the write bandwidth improvement is 8.6%, on average. Our frequency tuning exhibits higher bandwidth improvements in cases of larger size requests. This is related to the amount of data (per request) that OpenExpress should move between the host and underlying memory backend. The small requests (4 KB∼16 KB) have a few system addresses that OpenExpress needs to be involved in parsing the PPR entries and perform DMA, which implies that there are more operations related to NVMe protocol management of SQ, CQ, doorbell, etc. To handle the large requests (32 KB), data transferring modules successfully remove the software intervention related to all PRP traversing, address parsing and DMA control. They are also well operated with other hardware automation modules in a pipeline manner, which can improve the overall performance by 17.1%. We observe that the performance improvement of reads are more significant (Figure 5a). The read bandwidth improvement for 4 KB, 8 KB, 16 KB and 32 KB-sized requests is 5%, 6.1%, 16.4% and 59.4%, on average, respectively. All NVMe-related traffic for the reads are not congested as much as that of the writes thanks to the dual-simplex design of PCIe. Specifically, as the operations of queue dispatching modules are mostly engaged with PCIe inbound links, data transferring modules can write up data to the host system memory in serving the read requests via PCIe outbound links, in parallel. Because of this, our hardware automation becomes more promising to serve the large block-sized read requests than the writes.

Real workload. We compare the bandwidth and latency of OpenExpress with those of Optane SSD by executing diverse workloads, and the results are shown in Figures 5c and 5d, respectively. For better understanding, we decompose the bandwidth and latency for each workload, which are shown in Figures 5e and 5f, respectively. The bandwidth and latency of OpenExpress are better than that of Optane SSD by 76.3% and 68.6%, on average, respectively. Since our frequency turning gives better performance on reads, the performance on read-intensive workloads (DevDiv and Server) is better than write-intensive workloads (24HR, FIU and TPCE). In particular, OpenExpress achieves more than 4 GB/s on DevDiv workload execution, which is 111.5% higher than the performance that Optane SSD provides (2.1 GB/s).

To see each of device bandwidth behaviors, Figure 5g shows the time series analysis of read-only and write-only bandwidth for DevDiv and 24HR workloads in a certain amount of I/O processing periods. As reads are dominant on DevDiv, the read-only bandwidth of OpenExpress is 27.1% better than Optane SSD. In contrast, the write-only bandwidth of OpenExpress for 24HR shows a similar or a little bit worse characteristic, compared to Optane SSD. We conjecture that Optane SSD buffers write requests into its internal DRAM, and thus, FPGA-based automation makes the performance difference between OpenExpress and Optane SSD (all logic is built on an ASIC). For both read and write bandwidth, the trends of dynamic performance curve for OpenExpress and Optane SSD are similar to each other.

5 Discussion, Related Work and Future Work

Related work. There are only a handful of frameworks that enable NVM-related storage systems research. OpenSSD [27, 28] provides flash software, which can be modified by users based on their research demands. We believe that OpenSSD offers yet an excellent opportunity to examine the diversity in flash management, but it only supports low speed SATA and non-standard interfaces. Unfortunately, their low performance (<350MB/s [33]) is not suitable for conducting research related with high-speed NVM technologies. In addition, OpenSSD does not allow users to modify hardware design and architecture. Note that OpenSSD is also not a cost-free open-source platform; Jasmine and Cosmos require around $2K and $3.5K per-device license fees, respectively.

Dragon fire card (DFC) implements a customized FPGA design, but it is only used for back-end flash controllers. All other functionalities are managed by software and a heavy operating system, which executes on multiple CPU cores. DFC is not in a public domain yet and unfortunately available for only a small, closed research community. Note that, in addition to the lack of open licensing and limitations with respect to support for fast, NVM devices, all the above frameworks require software to manage the host communication and NVMe protocol in contrast to OpenExpress.

Limits. Our hardware automated logic can remove software involvement for NVMe management, thus, providing performance inline with the expected user case – a fast PCIe storage card for NVM research. However, the automated hardware modules may require more technical efforts than what software-based NVMe utilities need to pay for a change, in cases where one needs to either apply the hardware logic to different types of FPGA technologies, or employ additional hardware/software logic. While OpenExpress implements most critical modules on both read and write paths, it requires users to modify the hardware logic to add new NVMe features and other commands for good measure. Even with these limits, we believe that OpenExpress is worthwhile to explore, as third-part NVMe IP cores are hard nut to crack to access for research purposes because of their expensive per-month unitary prices and restrictions to modify.

Future work and use cases. As an on-going research, we are updating the existing Linux storage stack and memory subsystem modules to exploit the true performance, exposed by different types of emerging NVMs. We are also fully updating OpenExpress to have a real NVM storage backend rather than DRAM emulation for a device-level study. This project requires a set of extra RTL designs and modifications to remove all firmware executions from the internal I/O path of an existing SSD. It also needs new types of NVM controllers the corresponding and physical layers to correctly handle the NVM backend complex with a low power. Lastly, we leverage a part of OpenExpress to i) build disaggregated pooled memory over a cache coherent interconnect for accelerators and ii) process data near storage class memory, which requires different computational IP cores for such new interface and in-storage processing acceleration.

6 Acknowledge

The author thanks Raja Appuswamy and Heiner Litz for shepherding this paper. The author also thanks anonymous reviewers for their constructive feedback as well as MemRay for their sample donations and technical support. This work is supported by MemRay grant (G01190170) and performed as a part of its open NVMe interface research project.

7 Conclusion

We propose an open-source hardware research platform, called OpenExpress. OpenExpress is an NVMe host accelerator IP for the integration with an easy-to-access FPGA design. We prototype OpenExpress on a commercially available Xilinx FPGA board and optimize all the logic modules to operate a high frequency. Using a thorough evaluation, we demonstrate that OpenExpress, with a maximum bandwidth of 7 GB/s, is an open research platform that can further storage system research related to fast NVM devices.

References

[1] ARM. AMBA AXI and ACE protocol specification. https://static.docs.arm.com/ihi0022/d/IHI0022D_amba_axi_protocol_spec.pdf.

[2] AXBOE, J. Flexible I/O tester. https://github.com/axboe/fio.

[3] BITTWARE. Xilinx ultrascale 3/4-length pcie board. http://www.bittware.com/wp-content/uploads/datasheets/ds-xusp3r.pdf.

[4] CHOI, W., JUNG, M., AND KANDEMIR, M. Parallelizing garbage collection with I/O to improve flash resource utilization. In *the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2018).

[5] DAYAN, N., KJÆR SVENDSEN, M., BJØRLING, M., BONNET, P., AND BOUGANIM, L. Eagletree: exploring the design space of ssd-based algorithms. In *Proceedings of the VLDB Endowment* (2013), vol. 6, ACM, pp. 1290–1293.

[6] ELYASI, N., ARJOMAND, M., SIVASUBRAMANIAM, A., KANDEMIR, M. T., DAS, C. R., AND JUNG, M. Exploiting intra-request slack to improve ssd performance. In *the 27th International Conference on Architectural Support for Programming anguages and Operating Systems (ASPLOS)* (2017).

[7] EPOSTAR-ELECTRONICS. Meissa nvme. http://www.epostar-elec.com/products_IP_Cores_Meissa.html.

[8] EVERSPIN. MR2A16A. https://www.everspin.com/supportdocs/all.

[9] FADU. Fadu annapurna ssd controller. http://fadutec.com/wp-content/uploads/2019/08/FADU-Annapurna-Brief-073019-WEB.pdf.

[10] FLASHMEMORYSUMMIT. PCIe/NVMe in mobile devices. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150811_S101C_Baram.pdf.

[11] GOUK, D., KWON, M., ZHANG, J., KOH, S., CHOI, W., KIM, N. S., KANDEMIR, M., AND JUNG, M. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *51th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).

[12] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing (ICS)* (2011).

[13] INTEL. INTEL® OPTANE™ SSD DC P4800X SERIES. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf.

[14] INTEL. PF29F64G08LCMFS.

[15] INTELLIPROP. NVMe target core. http://intelliprop.com/hardware-storage-design/ip-cores/nvme-target-ip-core-IPC-NV163-DT.htm.

[16] IP-MAKER. NVMe product overview. https://www.ip-maker.com/index.php?option=com_phocadownload&view=category&download=34:nvme_product_overview&id=6:nvm.

[17] JIN, Y. T., AHN, S., AND LEE, S. Performance analysis of nvme ssd-based all-flash array systems. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018), IEEE.

[18] JUNG, M., AND KANDEMIR, M. Middleware - firmware cooperation for high-speed solid state drives. In *Middleware '12* (2012).

[19] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. Simplessd: modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters 17*, 1 (2017), 37–41.

[20] KIM, J., LEE, D., AND NOH, S. H. Towards SLO complying ssds through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), USENIX Association.

[21] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Reflex: Remote flash = local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17.

[22] LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., BJØRLING, M., AND GUNAWI, H. S. The case of FEMU: cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[23] MICRON. P8P parallel phase change memory. https://media.digikey.com/pdf/Data%20Sheets/Micron%20Technology%20Inc%20PDFs/NP8P128Ax60E_Rev_K.pdf.

[24] MICROSEMI. Flashtec NVMe Controllers. https://www.microsemi.com/product-directory/storage/3687-flashtec-nvme-controllers.

[25] Murugan, M., and Du, D. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011).

[26] NVM Express Inc. NVM express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2017.

[27] OPENSSDTEAM. Cosmos openssd platforms. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Platform.

[28] OPENSSDTEAM. Jasmine openssd platforms. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[29] Park, J. K., Lee, J.-y., and Noh, S. H. Divided disk cache and ssd ftl for improving performance in storage. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE 17*, 1 (2017), 15–22.

[30] PCI-SIG. PCI express base specification revision 3.1a, 2015.

[31] SK hynix. PCIe NVMe controller firmware and drivers. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150813_FJ31_Parepalli.pdf.

[32] SNIA. Block I/O traces. http://iotta.snia.org/traces/list/BlockIO.

[33] Song, Y. H., Jung, S., Lee, S.-W., and Kim, J.-S. A pcie-based open source ssd platform. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2014/20140807_301B_Song.pdf.

[34] Tavakkol, A., Gomez-Luna, J., Sadrosadati, M., Ghose, S., and Mutlu, O. MQSim: A framework for enabling realistic studies of modern multi-queue ssd devices. In *16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[35] Xilinx. AXI Bridge for PCI Express. https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie3/v3_0/pg194-axi-bridge-pcie-gen3.pdf.

[36] Xilinx. Ultrascale architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[37] Yoo, J., Won, Y., Hwang, J., Kang, S., Choi, J., Yoon, S., and Cha, J. Vssim: Virtual machine based ssd simulator. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)* (2013).

[38] Zhang, J., Kwon, M., Gouk, D., Koh, S., Lee, C., Alian, M., Chun, M., Kandemir, M. T., Kim, N. S., Kim, J., and Jung, M. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[39] Zhang, J., Kwon, M., Swift, M., and Jung, M. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020), USENIX Association.

[40] Zhang, Y., Prasath Arulraj, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)* (2012).

Fast Software Cache Design for Network Appliances

Dong Zhou[♣][*], Huacheng Yu[◇], Michael Kaminsky[♡], David G. Andersen[♡][♠]
Tsinghua University[♣], Princeton University[◇], BrdgAI[♡], Carnegie Mellon University[♠]

Abstract

The high packet rates handled by network appliances and similar software-based packet processing applications place a challenging load on caches such as flow caches. In these environments, both hit rate and cache hit latency are critical to throughput. Much recent work, however, has focused exclusively on one of these two desiderata, missing opportunities to further improve overall system throughput. This paper introduces *Bounded Linear Probing* (*BLP*), a new cache design optimized for network appliances. BLP works well across different workloads and cache sizes by balancing between hit rate and lookup latency. To accompany BLP, we also present a new, lightweight cache eviction policy called Probabilistic Bubble LRU that achieves near-optimal cache hit rate (assuming the algorithm is offline) without using any extra space. We make three main contributions: a theoretical analysis of BLP, a comparison between existing and proposed cache designs using microbenchmarks, and an end-to-end evaluation of BLP in the popular Open vSwitch (OvS) system. Our end-to-end experiments show that BLP is effective in practice: replacing the microflow cache in OvS with BLP improves throughput by up to 15%.

1 Introduction

Network virtualization is a core infrastructure component for cloud computing. In virtualized networks, virtual switches route packets between virtual machines (VMs) and between VMs and the outside world. Like the VMs themselves, the virtual switch resides in the hypervisor. The high speed of modern NICs—40Gb/s, 100Gb/s, and even 200Gb/s [2]—makes virtual switches a critical network performance bottleneck.

Many software-based network systems, such as appliances, middleboxes, packet analytic frameworks, and virtual switches, rely on fast flow caches to achieve good average-case performance [10, 38]. These environments impose challenging—and, indeed, somewhat contradictory—requirements upon the caches they use. First, of course, they benefit from high hit rates. But, either to avoid wasting memory or to fit in faster levels of the CPU cache, they also strive to be compact. In addition, because of the high rates at which packet-centric systems operate, the flow cache lookups must have extremely low latency.

These competing requirements place such systems in an interesting middle ground compared to much of the prior work, which usually fall into one of the two extremes. Higher-level caching systems such as web caches and memcached often adopt comparatively expensive cache designs and replacement algorithms to maximize hit rate [9, 26]. On the other hand, CPU caches have such tight timing requirements that they use very simple set associative designs that sacrifice hit rate for extremely low access time measured in clock cycles.

In this paper, we present the design, theoretical analysis, and empirical evaluation of a new cache design called *Bounded Linear Probing*, or BLP, that provides higher cache hit rates than simple set-associative designs, while remaining fast and hardware-friendly. BLP achieves low latency by ensuring *purely local access* to the cache data structure: Look-ups require a single read that spans at most two consecutive CPU cache lines. At the same time, BLP allows *non-local propagation* of full buckets. A basic set-associative cache provides only one location for a given set of objects. BLP allows those objects to creep into later bins, and over repeated inserts and evictions, this property allows high-occupancy bins to shift some of their load to nearby, less-occupied bins. To better serve skewed workloads, we accompany it with a cache eviction algorithm called *Probabilistic Bubble LRU*, or PBLRU, that fulfills the same design goals as BLP: It requires no extra space, adds little latency overhead and achieves near-optimal cache hit rate.

BLP is a simple and effective design for performance-critical software caches; despite its simplicity, we believe it to be a novel design point in the space of "cache table" designs, and provide a theoretical analysis of why it provides an improved hit rate over basic set-associative designs that access the same number of elements. The result is a design that performs nearly as well as the fastest set-associative designs, with hit rates that are closer to that of more advanced, yet expensive, designs such as cuckoo or hopscotch-based caches. We validate these results empirically using both microbenchmarks and by incorporating BLP into Open vSwitch [30], the most popular virtual switch, which is widely used in production. Replacing the microflow cache in OvS with BLP improves throughput by up to 15%: Its lookup latency is about 10 clock cycles longer than that of the basic set-associative design, but BLP's increased cache hit rate more than compensates for the higher latency. In contrast, many of the more expensive

[*] work started while at Carnegie Mellon University

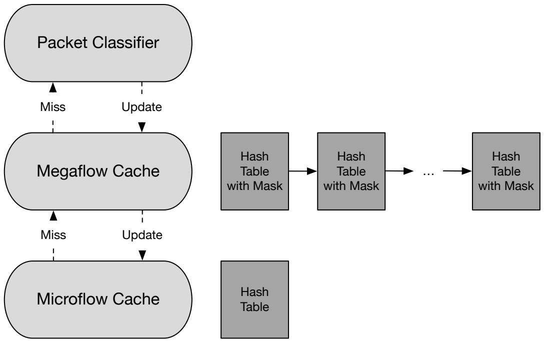


Figure 1: Flow Caching Hierarchy in Open vSwitch

cache designs that can achieve high cache hit rates do not justify their huge latency penalties. Our new cache eviction algorithm PBLRU further improves the throughput by up to 10% even if the workload is only modestly skewed.

2 Flow Caching in Open vSwitch

Open vSwitch achieves high performance through extensive flow caching. Open vSwitch's caching hierarchy consists of three layers: a microflow cache, a megaflow cache, and a caching-aware packet classifier, as illustrated in Figure 1.

The first cache that a packet encounters in OvS is the *microflow cache*, which caches forwarding decisions for each transport connection (or *microflow*). The microflow cache is a hash table that maps microflows to OpenFlow flows if there is an exact match using all the packet header fields. If the packet misses in the microflow cache, then OvS does a lookup in its *megaflow cache*. This cache supports wildcard matching but does not use flow priorities. The megaflow cache is a set of n hash tables, each with a unique wildcard mask. For each hash table, the lookup key is the packet header after applying the mask associated with the table. These hash tables are reactively created and populated by the packet classifier. Because looking up a packet in the megaflow cache searches all n hash tables, it is more expensive than a microflow cache lookup. Therefore, the cache hit rate of the microflow cache (the first cache) is critical to the performance of OvS.

The key observation that inspired our work is that although a microflow cache miss is expensive, it is not *immensely* more expensive than a microflow hit. In typical deployments, where the average number of hash table searches per megaflow lookup is small (as noted in Section 7.2 of Pfaff et al. [30]), the microflow miss penalty is only hundreds to thousands of cycles on modern server CPUs. Hence, making the correct tradeoff between cache hit rate and lookup latency is crucial to the system throughput. In contrast, much of the previous work on software cache designs focuses primarily on improving cache hit rate [9, 14, 18, 13]. In the situations studied by previous work, optimizing for hit rate makes sense: the cache misses in these systems were much more expensive than a hit because they often involved querying very slow backend services such as a database. The cache hit rate, therefore, determines not only the throughput, but also end-to-end request latency [14, 27].

The rest of this paper uses the OvS microflow cache as a case study to analyze and evaluate various design options and demonstrate the effectiveness of our new caching algorithm, Bounded Linear Probing (BLP). We show how BLP can balance cache hit rate (and thus miss penalty) and lookup latency to improve the throughput of Open vSwitch compared to alternate designs.

3 Background and Related Work

3.1 Network Packet and Flow Caching

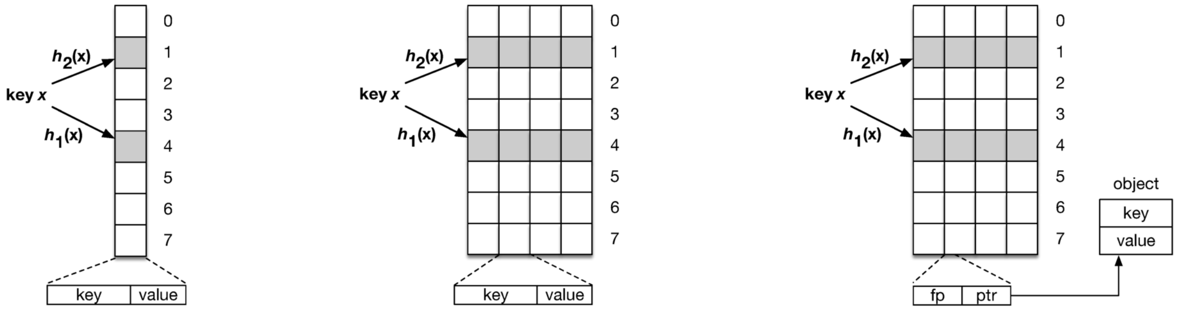
Caching is a common and effective technique for speeding up network packet processing; existing solutions include hardware-based [12, 38, 29] and software-based [10] approaches. Many early hardware routers used flow caching to achieve fast average-case performance. In the modern era, most hardware routers and switches have moved to more costly, but guaranteed-performance designs, such as TCAMs, to be able to provide their maximum forwarding rate under arbitrary (and possibly malicious) traffic. Software switches, however, broadly retain a cache-based design [3, 36].

3.2 Hash Table Options For Caching

Caching is typically managed using a hash table as its basic data structure, but unlike the "full" problem of a general hash table, caches gain an extra degree of freedom: By definition, they do not need to store all possible keys and may choose to evict an existing item.

One of the contributions of this paper is to explore the tradeoff between the cache's hit rate and the lookup/insertion cost imposed by its hash table structure. To illustrate this tradeoff, we begin in Section 4 by describing points that operate at two extremes of the spectrum: First, a basic set-associative cache, in which an item can be stored only in one of m different slots shared by all other items that hash to the same bucket (row) of the hash table. This design is fast but achieves a relatively low hit rate. Next, we introduce two more advanced cache designs that incorporate ideas from cuckoo and hopscotch hashing, which can achieve much higher table occupancy (and thus hit rates), but at the cost of more expensive inserts and lookups. In the rest of this section, we present prior work on fast caches, including a brief introduction to cuckoo and hopscotch hashing.

Cuckoo and hopscotch hashing Cuckoo [28] and hopscotch [21] hashing both aim to achieve high table occupancy (upwards of 90%) in an "open-addressed" hash table design, i.e., one that does not need to use linked lists to store data items. The pointer chasing of a linked-list design adds substantial lookup latency, and the pointers themselves can add substantial memory overhead, especially when the entries in the table are small, which is the case for flow caches.



(a) Basic cuckoo hash table

(b) 4-way set-assoc. cuckoo hash table

(c) Partial-key cuckoo hashing

Figure 2: Cuckoo Hashing

Both designs share a theoretical basis that allowing items to occupy a small number of slots that are "not too close" to each other allows buckets that would otherwise be too full to spill their contents recursively into other, hopefully less-full buckets. (Recall that when throwing n balls into n bins, the maximally loaded bin will have in the range of $(2 + o(1)) \times \frac{\log n}{\log \log n}$ elements. With 1 million entries in a table, this would result in more than 10 elements falling into the most-loaded bin. Traditional linear probing approaches handle this by keeping the table occupancy under 50%, which conflicts with the desire for high table occupancy.)

Cuckoo hashing Cuckoo hashing [28] maps each key x to two candidate buckets using two hash functions (two is a common choice, but using a larger number of hash functions is also possible). Each key is stored in one of these candidate buckets. Figure 2a shows a basic cuckoo hash table with eight buckets, where key x maps to bucket 4 by $h_1(x)$ and bucket 2 by $h_2(x)$. To lookup a key in a cuckoo hash table, we need to check both candidate buckets to see if the key resides in either bucket. Inserting a new key might relocate an existing key to its alternate bucket. In its basic form, cuckoo hashing provides an expected table load factor of 50%. By using 4-way or higher set-associative buckets, as shown in Figure 2b, the table space utilization increases to over 90% [17].

Partial-key cuckoo hashing [24, 18], illustrated in Figure 2c, extends the basic cuckoo hashing by storing in the table itself only a short hash of the key along with a pointer to the value. This short hash, or *fingerprint*, serves two purposes. First, during lookups, it eliminates the necessity of retrieving full keys unless the fingerprints match. Although false positives could happen, the probability is considerably low. For example, with 1-byte fingerprints, the chance of fingerprint-collision is less than 0.4% ($1/2^8$). Second, fingerprints are enough to derive the alternative bucket for a key, thereby allowing insertion to complete without retrieving the full keys. Replacing full keys with fingerprints substantially improves the memory efficiency and the performance of cuckoo hash tables.

Recent work proposes using a partial-key cuckoo hashing–based key/value cache to improve the performance of OvS (Cuckoo Distributor [37]). For each microflow, it stores a 16-bit fingerprint of the full packet header (key) along with a 16-bit megaflow index (value) in the table. This approach achieves high cache hit rate, but, as we will show in Section 7, the need to access two unrelated cache lines is detrimental to its lookup latency, resulting in lower overall performance.

Full key versus fingerprint The decision to store full keys versus smaller fingerprints goes beyond cuckoo hashing. Open vSwitch itself has two implementations of the microflow cache. The first implementation, called the exact-match cache or EMC [1], matches all the packet header fields to avoid false positives. Because each entry in EMC takes more than 550 bytes, it can only cache a small number of microflows. As of OvS v2.10.1, the default EMC configuration has a capacity of 8192 entries. This limitation causes severe performance drops when the number of active flows is large, as most packets will miss the microflow cache and trigger megaflow lookups [37].

To overcome this shortcoming, OvS introduced a second implementation, called the signature-match cache or SMC [4]. The signature-match cache is a 4-way set-associative cache that maps 16-bit fingerprints, instead of full packet headers, to megaflows. The megaflow entry can then be used for verification of the full flow header. This simple cache design has fast lookup latency; however, as our analysis demonstrates in Section 4, it suffers from low cache hit rate. This is the flow cache upon which the rest of our paper focuses.

Hopscotch Hashing Hopscotch hashing [21] is an open addressed hashing scheme that uses multi-phased probing and displacement to resolve hash collisions. In hopscotch hashing, a key hashed to a bucket will always be stored in that bucket, or in one of the next $H-1$ buckets, where H is a constant. In other words, a bucket and its next $H-1$ neighbors form a *virtual bucket* with H slots. To accelerate key lookup and insertion, each bucket maintains an H-bit bitmap, indicating which of the H slots in its virtual bucket contain keys that are hashed to the bucket. These bitmaps are updated during key displacement.

Because all the candidate buckets of a key are contiguous in memory, hopscotch hashing has good cache locality. Our cache design, BLP, resembles hopscotch hashing in the sense that it allows a key to be placed in one of the H buckets starting from the one it is hashed to.

3.3 Hardware Cache Designs

Although the focus of this work is on software caches, there are many parallels to related work on hardware caches.

Cache hit rate versus lookup latency Balancing the cache hit rate and lookup latency has been studied in the context of DRAM hardware caches. Alloy Cache [31], for example, improved performance over prior work by reducing the hit latency, even though doing so slightly reduced the hit rate.

Set-associative caches Hardware caches are often organized into *rows* (i.e., buckets) and *ways* (i.e., slots). An m-way set-associative cache uses a subset of the address bits to index into a row; the cache block (cache line) can be stored in one of the row's m ways. To balance the load across rows, researchers have proposed using a hash of the block address as the index [23] as is commonly done in software-based hash tables and caches.

Skewed-associative caches and cuckoo-like cache designs Skewed-associative caches [35] extend this idea and allow each way to be indexed with a different hash function. In an m-way skewed-associative cache, a cache block B could be stored in row $h_i(B)$ for way i, for $0 \le i < m$.

Inspired by cuckoo hashing, zcache [34] is an extension of skewed-associative caching. Instead of replacing one of the m existing blocks on a cache miss, it performs a breadth-first search to find additional eviction candidates. After picking a victim entry, it relocates blocks on the cuckoo path to accommodate the new block. These designs are not well-suited for high speed, low latency software caches for packet processing, as they require several cache line reads per lookup.

3.4 Cache Design and Eviction Policy

A large amount of prior work on caching [7, 20, 32, 9, 8, 14, 13] focuses on cache eviction policies. Improved policies, ranging from LRU and LFU to modern alternatives such as LHD [9], increase cache hit rate under skewed workload distributions by biasing eviction towards likely less-useful candidates.

The majority of prior cache eviction algorithms require additional tracking metadata to implement their eviction policies. In contrast, our new algorithm, PBLRU, adds no space overhead. The most related work to our algorithm is an earlier paper by Zhang and Xue [39] that explores the same *bubbling* idea. We discuss the differences between PBLRU and their algorithm, DC-Bubble, in Section 5.3.

4 Design and Analysis

We begin by presenting two baseline cache designs, a set-associative option and a "cuckoo-like" option, and analyze their expected hit rates. We then introduce bounded linear probing and its analysis using the same framework.

To understand the expected hit rate, we assume that the working set is fixed, and that each lookup key is drawn *uniformly* at random from that working set. We only analyze uniform distributions in this section, for the following two reasons: a) prior works studied caching performance on uniformly-distributed workloads [25] and b) the expected hit rate under the uniform distribution is easier to analyze, yet it provides a *lower bound* on the hit rate under any other distribution (see Appendix C for a formal argument). We use α to denote the ratio of the working set size to the number of entries in the cache table, which we call the *oversubscription factor*. When $\alpha < 1$, the cache has more capacity than there are items in the working set. We determine hit rate in terms of α, and then provide numerical interpretations for some values observed in the OvS workloads, such as $\alpha = 0.95$. In Section 7, we show empirical hit rate curves for real implementations across a range of α values.

All of the designs we evaluate use some amount of set-associativity. The caches are partitioned into n buckets, each containing m entries. To determine if an item is in a bucket, the implementation examines whether it is stored in *any* of the entries in the bucket. The table contains a total of $n \times m$ entries, and the working set has size $\alpha \times n \times m$. To store a key-value pair (k, v), one hashes the key and determines in a table-specific way a set of (one or more) buckets that could hold the key, and stores both *fingerprint*(k) and v in an appropriate entry. In OvS, v is a pointer to a megaflow cache entry. Each design uses a different algorithm to decide which entry of the table will store a given pair.

4.1 Analytical Framework for Hit Rate

To analyze the expected hit rate of a cache design, it suffices to estimate the expected number of keys the cache could hold after a sufficiently long warm-up period. This is because, in our formulation, each cache access is uniformly random, so the cache hit rate is equal to the total number of cached keys divided by the size of the working set. Moreover, the number of keys stored in all the cache designs we evaluate never decreases with an increasing number of cache accesses, and it has a maximum value of $n \times m$. Therefore, it will eventually stop increasing. Denote the *final* number of occupied entries in bucket i by c_i. The probability of a cache hit is equal to

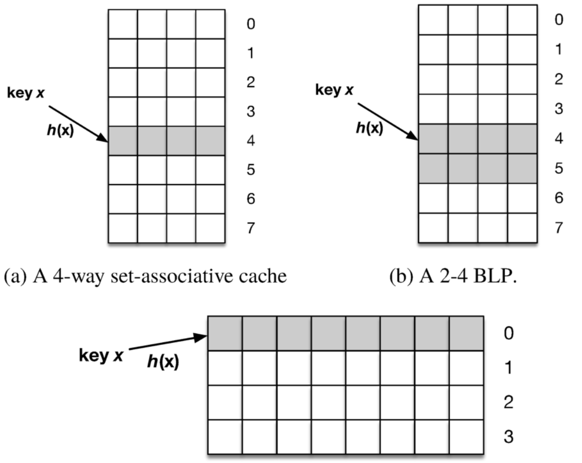
$$\frac{c_0 + \cdots + c_{n-1}}{\alpha n m}. \tag{1}$$

By symmetry, all c_i have the same expected value. Hence, by linearity of expectation, the expected cache hit rate is $\mathbb{E}[c_i]/(\alpha m)$. For each cache design, we describe how its hit rate is estimated from a high level, and leave all the details to appendices.

4.2 Set-associative Cache

We start with a simple design—a set-associative cache. In an m-way set-associative cache, each item is mapped to a bucket by a hash function h, and each bucket has m slots. Figure 3a and Figure 3c show 4-way and 8-way set-associative designs for identical-capacity caches respectively.

The lookup algorithm is straightforward. Given a key k, it checks bucket $h(k)$ for an entry containing *fingerprint*(k). Because it is possible for two different keys to have the same fingerprint, if the fingerprint matches, the full key is fetched to further verify it was indeed the correct one. The chance of fingerprint collision is small, roughly $m/2^{|\text{fingerprint}|}$. Insertion looks for an empty entry in bucket $h(k)$. If one exists, the new key-value pair goes there; if not, we evict a random entry in the bucket to make room for the new item.[1]



(a) A 4-way set-associative cache

(b) A 2-4 BLP.

(c) An 8-way set-associative

Figure 3: Set-associative Cache and BLP

We model the expected hit rate when performing a lookup on a uniformly random key from the working set using the standard balls-into-bins problem. We view each key in the working set as a ball and each bucket as a bin. Assuming the hash function is perfectly random, mapping the keys to the buckets can be viewed as placing αnm balls randomly into n bins, where each bin can hold at most m balls. When $\alpha \leq 1$, the data structure as a whole has sufficient *total* space for all balls, i.e., if the balls are placed evenly across the bins, no bin will overflow. However, a small fraction of the bins will still have more than m balls mapped to them. For set-associative caches, that means some keys (balls) will map to such buckets (bins) that are full, triggering a cache miss and an eviction.

Using the aforementioned analytical framework, our analysis shows that $\mathbb{E}[c_i] \approx m - \sum_{t=0}^{m}(m-t) \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!}$. (The full derivation of this result is listed in Appendix A.) Given $\alpha \approx 0.95$ (1,000,000 keys, 2^{20} cache entries), the cache hit rate of a 4-way set-associative cache is ~ 82% and the cache hit rate of an 8-way set-associative cache is ~ 88%. Both 4-way and 8-way set-associative caches have low lookup latency: assuming each item has a size of 32 bits (16-bit fingerprint + 16-bit value), each lookup only requires 1 cache line read. The downside, however, is their relatively low cache hit rates. As we will show in Section 7, low hit rates hurt throughput in many scenarios.

4.3 Cuckoo-lite

The second cache design, which we call *cuckoo-lite*, is very similar to cuckoo hashing. In an H-m cuckoo-lite cache, each bucket has m slots. Each key is mapped to H buckets by hash functions $h_1, h_2, \cdots, h_H$. A 2-4 cuckoo-lite cache shares the same structure as the 2-4 cuckoo hash table (Figure 2b).

Lookup in a cuckoo-lite cache is the same as lookup in a cuckoo hash table. Insertion, however, differs: To insert a key x to a 2-4 cuckoo-lite, we examine whether there is an empty slot in either of x's two candidate buckets. If there is, we insert x there; otherwise, we choose a random slot from the two buckets to replace with x. In other words, instead of searching for a cuckoo path to displace existing keys, cuckoo-lite simply *evicts* one of the existing keys to make room for the new key. This difference highlights the flexibility of a cache over a hash table. In the latter, every inserted key needs to be stored in the table. When there is no empty slot for the new key, the only options are to relocate an existing key via a sequence of key displacements, resize the table, or return an error. A cache, however, can evict arbitrary keys.

Although cuckoo-lite never relocates existing keys, it can achieve at least the same load factor as a cuckoo hash table after a sufficiently long warm-up period.

Following the analytical framework, the expected cache hit rate is at least $\frac{\text{load factor}}{\alpha}$. Prior work [17] estimated the achievable load of cuckoo hashing under various parameters using simulation. In particular, their experimental results showed that for 2-4 cuckoo hashing, 93% load is achievable 99% of the time.[2] Their results, when translated to a 2-4 cuckoo-lite cache, indicate that for $\alpha \leq 0.93$, the expected hit rate must be *at least* 0.99. Therefore, for $\alpha > 0.93$, the expected hit rate is at least $0.99 \times 0.93/\alpha$. Given $\alpha \approx 0.95$, 2-4 cuckoo-lite achieves close to 100% cache hit rate. The drawback, however, is the need to access two unrelated cache lines, which increases lookup latency.

4.4 Bounded Linear Probing (BLP)

We now present our cache design—Bounded Linear Probing, or BLP. The core idea behind BLP is simple: instead of restricting a key to only one bucket, or multiple unrelated buckets, BLP (like hopscotch hashing) can store it at one of the H buckets starting from the one to which it hashes. Unlike hopscotch hashing where each bucket maintains an H-bit bitmap for key displacement, BLP never relocates keys, and searches for all H buckets during the lookup, eliminating the need for per-bucket bitmaps. We call a BLP cache with m-way set-associative buckets, where the key can be in any of the $H-1$ subsequent buckets, an H-m BLP. Figure 3b demonstrates a 2-4 BLP. In this example, key x is mapped to

[1] We use random eviction in all cache designs, and discuss other eviction policies in Section 5.

[2] Their experiments are performed with 64K entries and a fixed amount of work per insertion.

Algorithm 1: `Lookup(`k`)` for 2-4 BLP

```
f = fingerprint(k);
i = h(k);
if bucket[i] has (f,ptr) then
    if ptr.key == k then
        return ptr.value;
j = (i+1) mod n;
if bucket[j] has (f,ptr) then
    if ptr.key == k then
        return ptr.value;
return not found;
```

Algorithm 2: `Insert(`k,v`)` for 2-4 BLP

```
f = fingerprint(k);
i = h(k);
if bucket[i] has an empty entry then
    add (f, ptr) to bucket[i];
    return Done;
j = (i+1) mod n;
if bucket[j] has an empty entry then
    add (f, ptr) to bucket[j];
    return Done;
select a random entry e from bucket[i] ∪ bucket[j];
replace e with (f,ptr);
return Done;
```

bucket 4 by hash function h. Therefore, x can be placed in either bucket 4 or bucket 5.

The lookup algorithm is straightforward. Given a key k, if any one of the H buckets contains *fingerprint*(k), the full key and value fetched via the pointer stored in the table. If the full key matches k, the algorithm returns the value, otherwise it returns "not found." See Algorithm 1 for the pseudocode of lookup for 2-4 BLP.

To insert a key-value pair (k,v) to a H-m BLP, the algorithm first checks if there is an empty slot in any one of the H buckets starting from $h(k)$. If so, the item is inserted to the empty slot; otherwise, the algorithm chooses an entry from the H buckets randomly (unless there is an entry with the same fingerprint, in which case the algorithm always chooses the matching entry), and replaces that entry with the new item. Algorithm 2 contains the simplified pseudocode for insertion for 2-4 BLP. Like cuckoo-lite, BLP does not relocate existing keys and reaches its maximum hit rate only after a warm-up period. Appendix D analyzes the relationship between the cache hit rate and the warm-up time in more details.

4.5 Why BLP might be better?

Intuitively, a 2-4 BLP cache should have higher cache hit rate than a 4-way or 8-way set associative cache, but lower hit rate than 2-4 cuckoo-lite. With 32-bit keys, a lookup in 2-4 BLP will read one or two *consecutive* cache lines; therefore, it has higher latency than 4-way and 8-way set-associative cache, but lower than 2-4 cuckoo-lite.

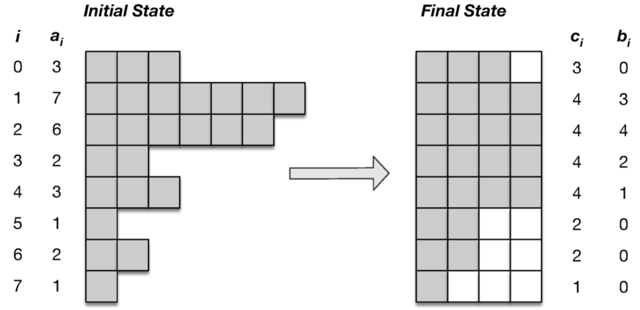


Figure 4: Analysis of BLP

Expected cache hit rate We use the following approach to analyze the expected hit rate of 2-4 BLP. We have two arrays a and b, where a_i is the number of keys from the working set that map to cache bucket i and b_i is the number of keys that spill from bucket i to bucket $i+1$ in the final state (after a sufficiently long warm-up period), for $i = 0, \dots, n-1$. See Figure 4 for an example.

In the final state, a total of $b_{i-1} + a_i$ keys are served by bucket i. When $b_{i-1} + a_i \le m$, bucket i has enough entries to store all these keys, and no keys will spill to bucket $i+1$, i.e., $b_i = 0$. If $m < b_{i-1} + a_i \le 2m$, bucket i can store m keys, and the extra $b_{i-1} + a_i - m$ will spill to the next bucket.[3] If $b_{i-1} + a_i > 2m$, then bucket i will store m keys, and m of the extra keys will spill to bucket $i+1$. Summarizing the above relations, we have the following equations for b_i and c_i:

$$b_i = \begin{cases} 0 & \text{if } b_{i-1} + a_i \le m, \\ m & \text{if } b_{i-1} + a_i \ge 2m, \\ b_{i-1} + a_i - m & \text{otherwise,} \end{cases} \tag{2}$$

$$\text{and } c_i = \min\{b_{i-1} + a_i, m\}, \tag{3}$$

where we assumed $b_0 = b_n$.

Analyzing b_i By Equation (3), the key to compute $\mathbb{E}[c_i]$ is to estimate the distribution of b_i, which is described by its probability densities $(p_0, \dots, p_m)$. We observed that b_i has almost identical distribution regardless of i, i.e., the same density vector $(p_0, \dots, p_m)$ describes all n distributions of $b_0, \dots, b_{n-1}$. Note that the above observation only concerns the marginal distribution of each b_i, and it does not assert how $b_0, \dots, b_{n-1}$ are jointly distributed. In fact, b_i and b_j for close i, j can be (very) correlated. We then observe that b_{i-1} is almost independent of a_i, the number of keys mapped to bucket i, and the distribution of a_i has a closed form. Therefore, we are able to derive a system of linear equations on the probability densities based on Equation (2). (The full derivation and solutions are listed in Appendix B.) For $\alpha = 0.95$ and $m = 4$, we calculated $\mathbb{E}[c_i] = 3.59$, and by Equation (1), the expected hit-rate is ~ 94%.

[3] Note that we only allow keys to spill to the next bucket. By definition, b_{i-1} is an integer between 0 and m, which implies that $b_{i-1} + a_i - m \le a_i$. Hence, there are sufficient keys that are mapped to bucket i to spill to the next bucket.

Design	Lookup Speed (cache line reads)	Hit Rate
4-way set-assoc	1	~ 82%
8-way set-assoc	1	~ 88%
2-4 cuckoo-lite	2 *unrelated*	~ 99%
2-4 BLP	1.5 *consecutive*	~ 94%

Table 1: Qualitative comparison of cache designs with α=0.95

Summary Table 1 summarizes 4-way set-associative, 8-way set-associative, 2-4 cuckoo-lite and 2-4 BLP caches in terms of lookup speed and estimated cache hit rate with $\alpha = 0.95$. 2-4 BLP increases the cache hit rate by 12% and 6% compared with 4-way and 8-way set associative caches, respectively, at a cost of 0.5 cache line reads on average.

5 Better Cache Replacement with Probabilistic Bubble LRU

The choice of cache eviction policy is largely orthogonal to the cache design. The cache design proposed in this paper, however, raises interesting and challenging requirements for the cache replacement policy. Like the cache design itself, we need the replacement policy to be fast and memory efficient; we also want a policy that can achieve near-optimal cache hit rate, especially in skewed workloads.

5.1 Traditional LRU approximations are too expensive

Perhaps the most commonly-deployed cache eviction policy is the least recently used (LRU) replacement. Traditional LRU suffers from several drawbacks, including high space overhead [9]. Even its approximations, such as CLOCK [15], requires maintaining 1-bit per key of state [18]. Although many applications can tolerate this amount of space overhead, introducing *any* extra cache management state is likely both to break the cache's careful memory alignment and to increase latency because of state updates.

5.2 Probabilistic Bubble LRU (PBLRU)

To address the aforementioned concerns, we propose a simple cache eviction algorithm that adds *no* extra space and produces near-optimal cache hit rate by combining both recency and frequency information. The core idea behind our algorithm is to assign different priorities to each slot in a bucket: the first slot has the highest priority, the second one has the second highest priority, so forth. After a cache hit, we *promote* the lookup key's priority by exchanging it with the key in the next highest priority slot (unless it is already stored in the first slot of the bucket). When we need to select a victim, we evict the key in the lowest priority slot (i.e., the last slot in a bucket). We call the process of promoting an entry *bubbling* and our algorithm *bubble LRU*. Bubbling effectively achieves the combination of LRU and LFU (least frequently used) without storing any extra information.

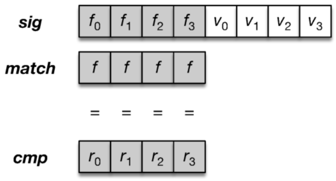


Figure 5: SIMD-optimized 4-way associative cache

Applying Bubble LRU to BLP Applying bubble LRU to set-associative caches is straightforward because each key is mapped to only one bucket. In BLP, however, each key could be placed in two or more consecutive buckets. To avoid introducing more complicated and expensive bucket selection logic, we simply pick a random bucket to apply bubbling.

Probabilistic Bubbling Although the basic bubble LRU does not use any extra space, it still requires a memory write on every lookup, which puts more pressure on the memory subsystem. To alleviate this issue, we adopt an optimization called *probabilistic bubbling*. With probabilistic bubbling, a cache hit does not always promote an entry's position in the bucket. Instead, we promote every n-th hit on average to reduce the number of memory writes. Intuitively, probabilistic bubbling should not hurt the cache hit rate because hot keys will still get promoted more often than cold keys. Our experiments in Section 7 demonstrate that PBLRU substantially improves the cache hit rate with modest overhead of lookup latency.

5.3 PBLRU and DC-Bubble

A non-probabilistic design of the bubble LRU with higher metadata tracking overhead called DC-Bubble was invented in 2009 by Zhang et al. [39]. Instead of always evicting the last entry in a bucket, DC-Bubble implements a more expensive replacement policy that requires updating an extra per-bucket bit on most accesses, which they use to track whether the last access to the same bucket is a hit or a miss. If the last access is a hit, then the last entry is replaced; otherwise, they evict the first entry, promote *all* the rest entries in the bucket, and put the new entry in the last slot.

PBLRU has two advantages over DC-Bubble. First, the authors of DC-Bubble designed their algorithm in the context of hardware cache where operations like setting a flag are cheap, but such operations incur non-negligible overhead in software caches. Second, it is non-trivial to convert their design to a probabilistic version. In our evaluation, we show that PBLRU has substantially lower latency and moderately higher cache hit rate than DC-Bubble.

6 Implementation

We have implemented all of the cache designs in C++. Like stock OvS's SMC (Section 3), we use 16-bit fingerprints and 16-bit values. Within each bucket, we store the fingerprints packed together followed by packed values. We applied several optimizations to improve the cache lookup performance:

SIMD-optimized Lookup To accelerate lookups, we use SIMD instructions to compare multiple fingerprints at the same time (similar to techniques used by Google's Swiss Tables [6]). Figure 5 shows how the lookup works in a 4-way set-associative cache. The stock OvS design does not use SIMD-accelerated reads for its microflow cache, so to ensure a fair basis for comparison, we implemented this optimization and use it as the baseline for comparison.

To search for fingerprint f in a bucket, we first duplicate it four times and store it in a 64-bit integer `match`. Then, we load the first 64 bits of the bucket into another 64-bit integer `sig`. We compare the packed 16-bit integers in `sig` and `match` for equality, storing the results in `cmp`. `cmp` consists of 4 16-bit integers r_0,r_1,r_2 and r_3, where r_i is 0xFFFF if $f_i = f$ and 0 otherwise. We can then count the number of trailing zeros in `cmp` to figure out which slot f matches in the bucket.

Lookup in an 8-way set-associative cache works similarly to the 4-way set-associative cache, but uses 128-bit integers instead of 64-bit integers. For 2-4 cuckoo-lite, because the eight candidate fingerprints are not consecutive, we have to first copy the fingerprints from two buckets into one 128-bit integer, then perform packed integer comparison.

SIMD-accelerated lookup in 2-4 BLP works as follows: The eight candidate fingerprints are not contiguous in memory (unlike cuckoo-lite), but are separated by the 64 packed value bits. Therefore, instead of *copying* fingerprints, we load *both* buckets into a wider 256-bit integer and mask off all the value bits. Eliminating the extra load instruction reduces the lookup latency by ~ 10% and makes BLP more SIMD-friendly than cuckoo-lite.

Buffer Bucket In 2-4 BLP, if the lookup key hashes to the last bucket of the table, both the first and the last bucket are searched. This corner case requires both a second cache line read and, more importantly, an extra branch. To avoid incurring branch prediction misses, we added a *buffer bucket* following the original cache table. This buffer bucket has minimal impact on the cache miss rate but improves lookup speed: When the lookup key hashes to the last bucket, and that bucket is full, the new key spills to the buffer bucket instead of wrapping around the table. At lookup time, we search both the last bucket and this buffer bucket, which avoids the branch misprediction and allows for processor prefetching[4]. One thing which worth mentioning is that this optimization is specific to BLP and does not work well with other cache designs — it breaks the alignment of the number of buckets (typically a power of 2). Moreover, the extra space given by this buffer bucket is negligible compared to the size of the cache. We therefore only apply the optimization to BLP.

Batched Lookup with Prefetching We use batched lookup with prefetching to overlap bucket computation with memory reads, which minimizes the impact of DRAM access latency. This technique is common in many existing packet processing applications and frameworks [40, 22, 11].

7 Evaluation

We present our evaluation top-down: We begin with a description of the experimental setup followed by a set of end-to-end benchmarks that compare the different cache table designs (described above) in the context of Open vSwitch. These results demonstrate the benefits and generality of BLP in a realistic packet processing application. Next, we use a set of microbenchmarks to understand more deeply the fundamental tradeoffs that each of the cache design brings to the table.

7.1 Experiment Setup

Our experiments are conducted on c220g2 instances from CloudLab [33]. Each of the instances is equipped with the following hardware:

Hardware	Description
CPU	2× Intel Xeon E5-2660v3 CPUs (2.60GHz)
DRAM	160 GiB DDR4 Memory
L3 Cache	2× 24 MiB
NIC	Intel X520 dual-port 10GbE

We also controlled for the following factors, which otherwise had noticeable effects on our results:

Random Number Generator Throughout the experiments, we use PCG-32 [5], a fast and statistically robust algorithm for our random number generation.[5]

Cache Warming As discussed in Section 4, 2-4 cuckoo-lite and 2-4-BLP do not displace keys. Instead, they depend purely on cache warming to reach the maximum hit rate. Therefore, in each experiment, we first warm the testing cache until it reaches a stable state, i.e., the cache hit rate stops increasing.

All experimental results reported below are the average of five runs. The variance was low, so we omit error bars from our graphs. Because the differences between many of the designs are small—in the range of 10% or so—while the absolute performance differences between a high cache hit rate (low alpha) and a low cache hit rate (high alpha) are relatively large, we deliberately choose not to start axes at 0; the graphs are "zoomed-in" to the regions of interest.

7.2 End-to-end Benchmarks

As a concrete end-to-end benchmark using an important application, we modified the microflow cache in Open vSwitch (v2.10.1) to use the various cache designs described above.

Open vSwitch was running on the a c220g2 instance with two 10Gb Ethernet ports, port 0 and port 1. To accurately

[4] Note that this optimization means that no keys will ever spill *into* the first bucket.

[5] The quality of the random number generator directly affects the cache hit rates. Unintended workload locality (i.e., back-to-back keys that hash to nearby buckets) produces higher than expected hit rates; poor random number generators exacerbate this effect. Earlier in the research process for this work, a bad, hand-crafted random number generator caused this issue.

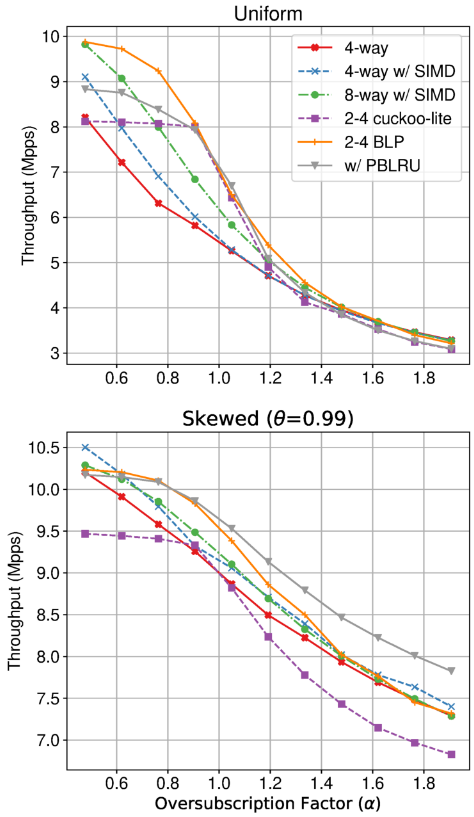


Figure 6: End-to-end Benchmark: Throughput

measure the throughput of Open vSwitch, we instruct the traffic generator to send synthetic network packets to port 0 of OvS and configure flows in OvS to have only one action: send packets out via port 1. This setup allows us to measure how fast OvS can process incoming packets by simply measuring the RX throughput on the traffic generator. We configured Open vSwitch to use one thread per port to handle network packets. On the traffic generator, we used MoonGen [16] to generate minimum-sized (64B) UDP packets. We configured it to use 4 TX threads, because a single thread could not saturate the 10Gb link.

Because public datasets are not readily available[6], we created a synthetic rule set and traffic patterns that creates roughly 20 hash tables in the megaflow cache, similar to the one used by Wang et al. [37]. By default, OvS's SMC has a total of 2^{20} entries (4 MiB), and we use the same cache size in our end-to-end benchmarks. We conducted our experiments with oversubscription factors between 0.5 and 2 because we believe they are the most plausible values in actual deployments – too low of oversubscription factor leads to underutilized caches while too high of oversubscription factor diminishes the effect of caching.

In our experimental result figures, "4-way" means a 4-way set associative cache (OvS's default SMC implementation without our SIMD optimizations), "4-way w/ SIMD" is the same cache design but with SIMD-optimized lookup. "8-way w/ SIMD," "2-4 cuckoo-lite," and "2-4 BLP" are 8-way set associative, 2-4 cuckoo-lite and 2-4 BLP caches respectively, each with their own SIMD-optimized lookups. "w/ PBLRU" is 2-4 BLP with PBLRU instead of random eviction. We use these notations throughout microbenchmarks and end-to-end benchmarks.

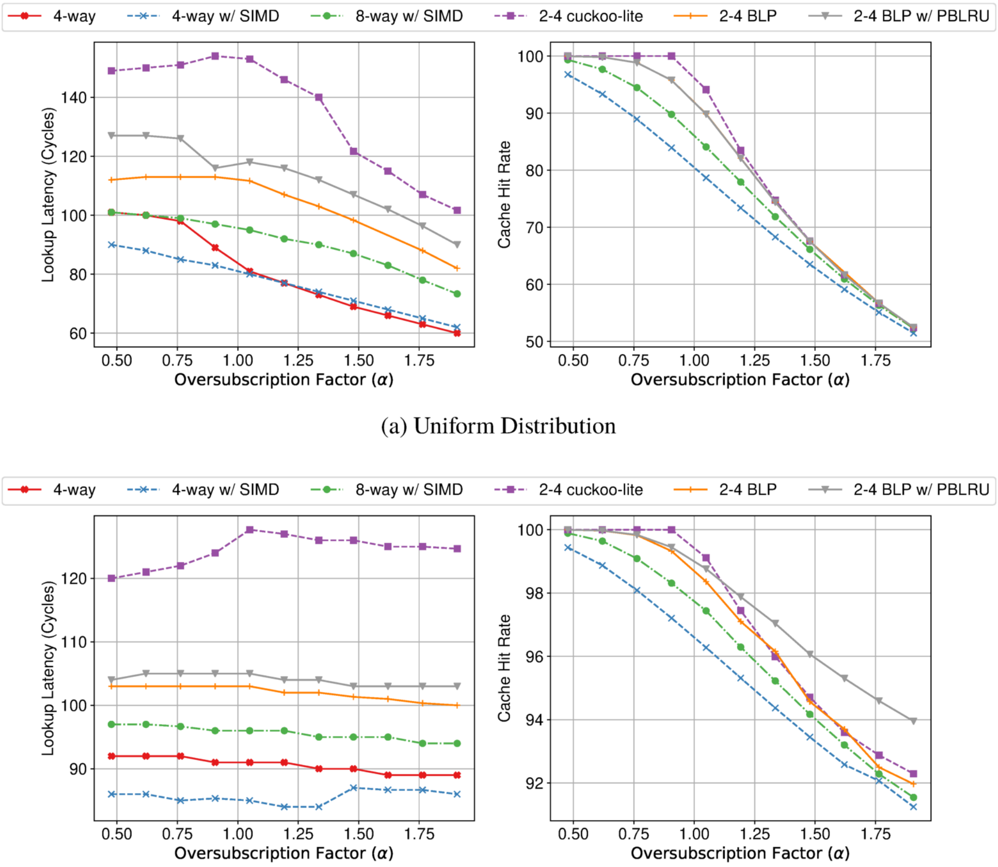
The overall throughput results are shown in Figure 6. Under a uniform workload, the higher hit rate achieved by BLP causes it to outperform alternative designs through a wide range of α values. Under a Zipf distribution of skewness 0.99 (denoted by θ) [19], BLP similarly outperforms in the middle range, losing to the lower-latency 4-way and 8-way SIMD lookups only at very low and very high oversubscription factors. PBLRU does not help in the uniform distribution, but improves the overall throughput in the skewed distribution. We explore the reasons for these results below.

Uniform Workload Figure 7a shows the microflow cache hit rate and microflow cache lookup latency that corresponding to the throughput numbers in Figure 6 (top). Despite 2-4 cuckoo-lite having a higher cache hit rate, and both 4-way and 8-way SIMD having lower lookup latency, 2-4 BLP is the clear winner. It achieves the highest throughput within the entire range of oversubscription factors, outperforming alternative designs by as much as 15%. 2-4 cuckoo-lite's marginal cache hit rate improvement is not enough to compensate for its higher lookup latency. Because PBLRU is unable to improve the cache hit rate in the uniform distribution, it slightly hurts the throughput.

Skewed Workload Figure 7b shows the latency and hit rate results for the Zipf distribution throughput in Figure 6 (bottom). As with the uniform distribution, the simpler 4-way and 8-way lookup designs have lower latency, and 2-4 cuckoo-lite has a mildly higher hit rate. But here too, 2-4 BLP achieves the highest throughput except when the oversubscription factor is very small (close to 0.5) or very large (close to 2). Even in such scenarios, its throughput is very close to the highest. PBLRU further improves cache hit rate, especially when the oversubscription factor is high and increases throughput by up to 7.5%. This is rather expected because when the oversubscription factor is low, using random eviction is enough to get hit rates that are close to 100%.

Summary When used to implement the microflow cache in OvS, 2-4 BLP has the best overall performance. It always achieves the highest, or close to the highest throughput across workloads. Compared to other designs, it improves the throughput by up to 15% with uniform workloads and up to 4% with skewed workloads. PBLRU further improves the throughput with skewed workloads by up to 7.5%.

[6] We contacted the lead author of Open vSwitch, who acknowledged this issue.



(a) Uniform Distribution

(b) Skewed Distribution

Figure 7: End-to-end Benchmark: Analysis

7.3 Microbenchmarks

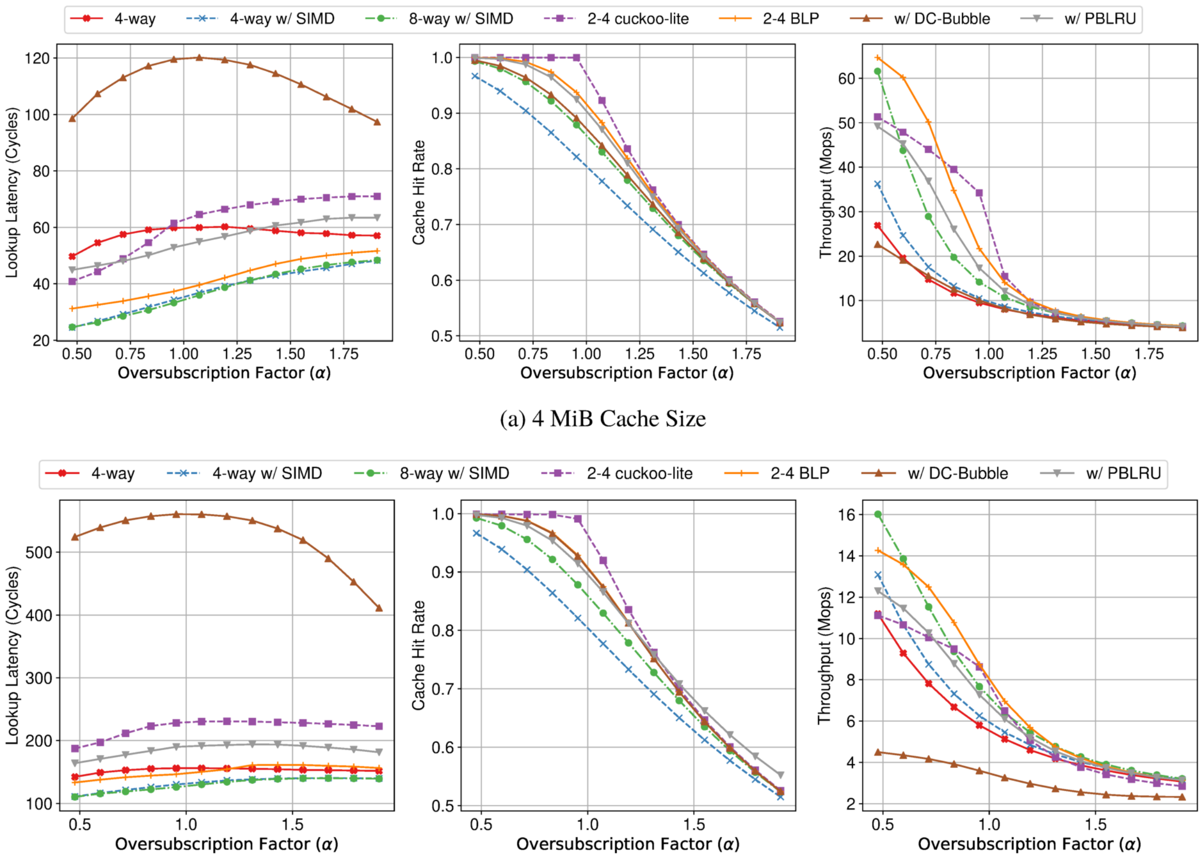
To understand how each cache design performs under different settings, we conducted a series of microbenchmarks outside of Open vSwitch. Each microbenchmark measures lookup latency, cache hit rate, and throughput with a *emulated* cache miss penalty of 1000 cycles. In these experiments, we emulate the cache miss penalty as a fixed cost instead of directing packets into the OvS megaflow lookup process, whose megaflow hash tables are searched can add substantial measurement noise. Because of this decision and because the microbenchmarks lack the instruction and data cache pressure of the full OvS system, the microbenchmark results differ slightly from the end-to-end numbers, but remain useful for exploring why and when the different cache designs excel. For microbenchmarks, we also implemented DC-Bubble [39] and compared it against PBLRU.

We evaluate two cache sizes to understand the performance when the table does and does not fit in the CPU's L3 cache. The workloads are pre-generated to minimize the impact of random number generation; each cache is fed exactly the same set of keys in the same order (for a given trial).

Uniform Workload, Small Cache Size We start by testing the cache designs for a small cache size (4 MiB) under uniform workloads. Figure 8a shows the results.

A few things stand out. As in the end-to-end benchmarks and analytical results, 2-4 cuckoo-lite has the highest cache hit rate for all oversubscription factors and 2-4 BLP has higher cache hit rate than 8-way. Second, 4-way w/ SIMD and 8-way have the lowest lookup latency, between 20 to 50 cycles per lookup. 2-4 BLP's latency is slightly higher and 2-4 cuckoo-lite is the slowest. Third, for oversubscription factors between 0.75 and 1.25, 2-4 cuckoo-lite achieves the highest throughput whereas 2-4 BLP achieves the highest for all other oversubscription factors. This result is because when the cache size is small, it is more likely to reside in the CPU's L3 cache (regardless of caching scheme). Therefore, although 2-4 cuckoo-lite issues one more memory read comparing with other approaches, its increased cache hit rate is enough to overcome the small lookup latency overhead. PBLRU is slower than 2-4 BLP, but much faster than DC-Bubble.

Takeaway When lookup latency overhead is small, higher cache hit rate produces higher throughput.



(a) 4 MiB Cache Size

(b) 64 MiB Cache Size

Figure 8: Micro-benchmark: Uniform Distribution

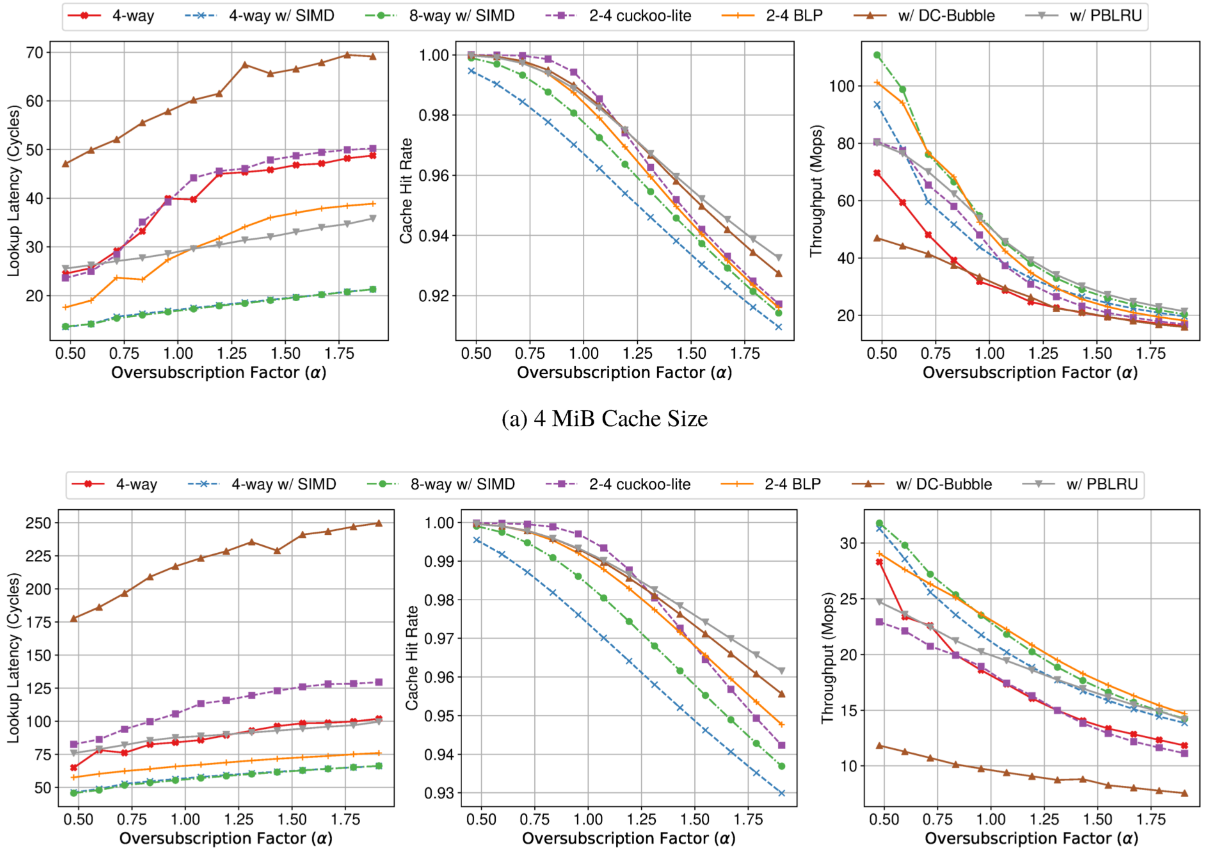
Uniform Workload, Large Cache Size In the second set of microbenchmarks, we increase the cache size to 64 MiB, which is larger than the CPU's L3 cache size. Comparing the results shown in Figure 8b with Figure 8a, the most notable difference is that lookup latencies are higher for all cache designs, especially 2-4 cuckoo-lite. Despite 2-4 cuckoo-lite's advantage in terms of cache hit rate, its overall throughput is lower than 2-4 BLP. Also, when oversubscription factor is small, 8-way outperforms 2-4 BLP because 8-way has a lower lookup latency than 2-4 BLP.

Takeaway When lookup latency overhead is large, lookup latency matters more than cache hit rate.

Skewed Workload The third and fourth set of experiments evaluate the cache designs with the same skewed workload (Zipf-distribution with θ=0.99) as in end-to-end benchmarks. Figures 9a and 9b depict the results for small and large cache sizes, respectively. Compared with uniform workloads, the relative order of 4-way w/ SIMD, 8-way, 2-4 cuckoo-lite and 2-4 BLP remains the same for both cache hit rate and lookup latency. However, across all the cache designs, cache hit rates are significantly increased and lookup latencies are significantly lower, which means that skewed workloads *favor* 4-way and 8-way set associative caches. As shown in these two figures, 8-way achieves the highest throughput. 2-4 BLP has much higher throughput than 2-4 cuckoo-lite (only 1-2% behind 8-way). With skewed workloads, PBLRU achieves higher cache hit rate, lower latency, and therefore higher overall throughput than DC-Bubble. Unlike in the end-to-end benchmarks, PBLRU has a lower throughput than 2-4 BLP in these microbenchmarks. This inversion is because the lookup latencies of both designs are substantially lower in the microbenchmarks, which amplifies the significance of the overhead introduced by PBLRU.

Takeaway With skewed workloads, 8-way and 2-4 BLP provide the highest throughput. PBLRU achieves higher cache hit rate than 2-4 BLP with modest overhead.

Summary As demonstrated by the micro-benchmarks above, 2-4 BLP balances cache hit rate and lookup latency across different cache sizes and workload skews. Compared with 4-way and 8-way set-associative caches, it has slightly higher lookup latency (1.5 v.s. 1 cache line reads / lookup), but much higher cache hit rate, especially with uniform workloads. Compared with 2-4 cuckoo-lite cache, 2-4 BLP replaces 2 *distant* cache line reads with one or two *consecutive* cache line reads, and therefore achieves much lower lookup latency at the cost of slightly lower cache hit rate.



(a) 4 MiB Cache Size

(b) 64 MiB Cache Size

Figure 9: Micro-benchmark: Skewed Distribution (θ=0.99)

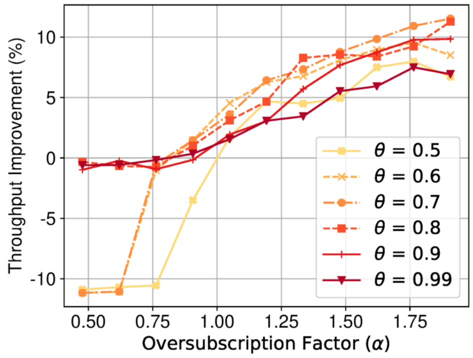


Figure 10: Throughput Improvement of PBLRU over 2-4 BLP

7.4 When should we use PBLRU?

Experiments above have shown that PBLRU outperforms random eviction in certain scenarios. To better understand when should we use PBLRU instead of random eviction, we compared the end-to-end throughput of 2-4 BLP with PBLRU against 2-4 BLP with random eviction using workloads with various skewnesses (θ). Higher θ means the workload is more skewed. Figure 10 illustrates the result.

Two things stand out. First, the performance improvement of PBLRU increases with the oversubscription factor. This is expected because when the oversubscription factor is low, the cache hit rate with random eviction is already high enough so that PBLRU's marginal cache hit rate improvement is not enough to overcome the additional lookup latency. Second, save for a few operating regimes (both skewness and oversubscription factor are low), PBLRU always provides equivalent or better throughput than the baseline.

8 Conclusion

Bounded Linear Probing (BLP) is a new cache design that introduces a new point in the "cache table" design space that balances between cache hit rate and lookup latency. BLP combines the speed of traditional set-associative caches with the high load factor (cache hit rate) and compact size of modern open-addressed hash tables. PBLRU is a new lightweight cache eviction algorithm that is fast and adds no space overhead. Our analysis and microbenchmarks show when each design offers advantages—for which workloads, data sizes, and miss penalties. Finally, we show that replacing the microflow cache in Open vSwitch can improve system throughput by up to 15% and PBLRU further improves the throughput by up to 10% when the workload is skewed.

References

[1] The open vswitch* exact-match cache. https://software.intel.com/en-us/articles/the-open-vswitch-exact-match-cache, .

[2] Mellanox ConnectX-6 product brief. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, .

[3] Open vSwitch, . http://www.openvswitch.org.

[4] [ovs-dev] [patch v5 1/2] dpif-netdev: Add smc cache after emc cache. https://mail.openvswitch.org/pipermail/ovs-dev/2018-July/349395.html, .

[5] Pcg, a family of better rarndom number generators. http://www.pcg-random.org, .

[6] Swiss tables and absl::hash. https://abseil.io/blog/20180927-swisstables, .

[7] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 421–432, Dec 2007. doi: 10.1109/MICRO.2007.42.

[8] N. Beckmann and D. Sanchez. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, March 2016. doi: 10.1109/HPCA.2016.7446067.

[9] N. Beckmann, H. Chen, and A. Cidon. Lhd: Improving cache hit rate by maximizing hit density. In *Proc. 15th USENIX NSDI*, Apr. 2018.

[10] F. Chang, W. chang Feng, and K. Li. Approximate caches for packet classification. IEEE INFOCOM, 2004.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), Aug. 2007. ISSN 0362-5915. doi: 10.1145/1272743.1272747. URL http://doi.acm.org/10.1145/1272743.1272747.

[12] T.-C. Chiueh and P. Pradhan. Cache memory design for network processors. In *6th International Symposium on High-Performance Computer Architecture (HPCA 2000)*, 2000.

[13] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, 2015.

[14] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334. USENIX Association, 2017. ISBN 978-1-931971-38-6.

[15] F. Corbato and M. I. O. T. C. P. MAC. *A Paging Experiment with the Multics System*. Defense Technical Information Center, 1968. URL http://books.google.com/books?id=5wDQNwAACAAJ.

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 15nd ACM SIGCOMM conference on Internet measurement*, IMC '15, 2015.

[17] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. Seventh Workshop on Distributed Data and Structures (WDAS'06)*, Jan. 2006.

[18] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384. USENIX, 2013. ISBN 978-1-931971-00-3.

[19] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, May 1994.

[20] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 107–116, June 2000. doi: 10.1145/339647.339660.

[21] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, 2008.

[22] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, May 2015.

[23] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 288–299, Feb 2004. doi: 10.1109/HPCA.2004.10015.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[26] Memcached. Memcached: A distributed memory object caching system. http://memcached.org/, 2011.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398. USENIX, 2013. ISBN 978-1-931971-00-3.

[28] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[29] C. Partridge and Others. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[30] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, 2015.

[31] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sramtags with a simple and practical design. In *Proc. ACM MICRO*, 2012.

[32] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555, June 2005. doi: 10.1109/ISCA.2005.52.

[33] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[34] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198, Dec 2010. doi: 10.1109/MICRO.2010.20.

[35] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178. ACM, 1993. ISBN 0-8186-3810-9. doi: 10.1145/165123.165152. URL http://doi.acm.org/10.1145/165123.165152.

[36] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow caching for high entropy packet fields. In *Proceedings of the third workshop on Hot topcis in software defined networking*, HotSDN '14, 2014.

[37] Y. Wang, T.-Y. C. Tai, R. Wang, S. Gobriel, J. Tseng, and J. Tsai. Optimizing open vswitch to support millions of flows. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM 2017)*, 2017.

[38] J. Xu, M. Singhal, and J. Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. IEEE INFOCOM, 2000.

[39] C. Zhang and B. Xue. Divide-and-conquer: A bubble replacement for low level caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 80–89. ACM, 2009. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542291. URL http://doi.acm.org/10.1145/1542275.1542291.

[40] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

A Expected Cache Hit Rate of Set-associative Caches

In a m-way set-associative cache with n buckets, for each bucket, the probability that there are exactly t keys mapped to it is

$$\binom{\alpha nm}{t} \cdot n^{-t} \cdot (1-1/n)^{\alpha nm - t}$$
$$= \frac{\alpha nm(\alpha nm-1)\cdots(\alpha nm-t+1)}{t! \cdot n^t} \cdot (1-1/n)^{\alpha nm - t}$$

which when $t \ll \alpha nm$, is approximately

$$\frac{(\alpha nm)^t}{t! \cdot n^t} \cdot (1-1/n)^{\alpha nm} = \frac{(\alpha m)^t}{t!} \cdot (1-1/n)^{\alpha nm}$$

which by the fact that $1 - \epsilon \approx e^{-\epsilon}$ for small ϵ, is approximately

$$\frac{(\alpha m)^t}{t!} \cdot e^{-\alpha nm/n} = \frac{(\alpha m)^t}{e^{\alpha m} t!}$$

If $t \le m$, then all t keys will be cached; otherwise, only m will be cached. Therefore, the expected number of keys that are cached in a bucket is approximately

$$\sum_{t=0}^{m} t \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!} + \sum_{t=m+1}^{\infty} m \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!},$$

which by the fact that $\sum_{t \ge 0} \frac{(\alpha m)^t}{e^{\alpha m} t!} = 1$, is equal to

$$= m - \sum_{t=0}^{m} (m-t) \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!},$$

Hence, the expected cache hit rate is $(m - \sum_{t=0}^{m}(m-t) \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!})/(\alpha m)$.

B Expected Cache Hit Rate of 2-4 BLP

Recall that a_i is the number of keys from the working set that map to cache bucket i and b_i is the number of keys spill from bucket i to $i+1$ after a sufficiently long warm-up period. For $j \ge 0$, we have:

$$\Pr[a_i = j] = \binom{\alpha nm}{j} \cdot n^{-j} \cdot (1-1/n)^{\alpha nm - j}.$$

By the law of total expectation and Equation (2) in Section 4.5, for $0 < l < m$, we have

$$p_l \approx \begin{cases} \sum_{j'=0}^{m} \left(\sum_{j=0}^{m-j'} \Pr[a_i = j]\right) \cdot p_{j'} & \text{if } l = 0, \\ \sum_{j=l}^{l+m} \Pr[a_i = j] \cdot p_{l+m-j} & \text{if } 0 < l < m, \end{cases}$$

and by the definition of probability distribution,

$$p_0 + \cdots + p_m = 1.$$

Solving the above system of linear equations for $(p_0, \ldots, p_m)$ with $m = 4$, $\alpha = 0.95$ gives us $p_0 = 0.37889778$, $p_1 = 0.15160669$, $p_2 = 0.14369602$, $p_3 = 0.12041777$ and $p_4 = 0.20538175$. By Equation (3), we get $E[c_i] = 3.59$.

C Expected Hit Rate of BLP under Non-Uniform Distributions

In the following, we prove that the expected hit rate obtained in Section 4.5 for the uniform distribution is always a lower bound for any other distribution. Fix any distribution over the working set S, let p_x be the probability of key x. Without loss of generality, we may assume $p_x > 0$ for all $x \in S$, since otherwise, we could simply remove all x with zero probability from the working set. Recall that c_i denotes the *final* number of occupied entries in bucket i. Observe that $c_0, \ldots, c_{n-1}$ are determined only by the hash function, i.e., how many keys are mapped to each bucket. They do not depend on the probability distribution of the keys (as long as all keys have non-zero probability), the distribution only affects how fast the final numbers are achieved.

After a sufficiently long warm-up period, all buckets achieved their final numbers of occupied entries. Now, consider all possible memory configurations *after* the warm-up. Further key lookups define a *Markov chain* over them, where the transition probability from memory configuration A to configuration B is the probability that A becomes B after one lookup. Observe that this Markov chain is *aperiodic* (i.e., there does not exist a $t > 1$ and a state A such that A can only go back to itself after steps of multiples of t). It is well-known that for any aperiodic Markov chain and any initial state, as the number of steps (key lookups) increases, the distribution of the state will approach *some* final stationary distribution (note that the stationary distribution may not be unique, hence the final stationary distribution may depend on the initial state). The *final* hit rate is computed from this stationary distribution and the distribution of the keys. More specifically, let q_x be the expectation, over a random hash function and random lookups in the warm-up period (which determine the Markov chain and the distribution of initial state), of the probability that key x is cached according to the final stationary distribution. Thus, the expected hit rate is equal to $\sum_{x\in S} p_x q_x$.

Next, by linearity of expectation, $\sum_{x\in S} q_x$ is equal to the expected total number of occupied entries in the data structure, $\mathbb{E}[c_i] \cdot n$. The key observation is that if $p_x \geq p_y$ then $q_x \geq q_y$, i.e., if a key is more likely to occur, then it has a higher probability to appear in the final stationary distribution, *over a random hash function and warm-up period.*[7] Let $S_{\text{high}} := \{x : p_x \geq 1/|S|\}$ be the set of keys that occur with at least the average probability and $S_{\text{low}} := \{x : p_x < 1/|S|\}$ be the set of keys that occur with probability lower than the average, and let $q := \min_{x\in S_{\text{high}}} q_x$. Hence, $q \geq q_x$ for all $x \in S_{\text{low}}$. We have

$$\begin{aligned}\sum_{x\in S} p_x q_x &= \sum_{x\in S} \frac{q_x}{|S|} + \sum_{x\in S}(p_x - 1/|S|)q_x \\ &= \frac{1}{|S|}\sum_{x\in S} q_x + \sum_{x\in S_{\text{high}}}(p_x - 1/|S|)q_x + \sum_{x\in S_{\text{low}}}(p_x - 1/|S|)q_x\end{aligned}$$

which by the fact that $p_x \geq 1/|S|$ and $q_x \geq q$ for $x \in S_{\text{high}}$, and the fact that $p_x < 1/|S|$ and $q_x \leq q$ for $x \in S_{\text{low}}$, is at least

$$\begin{aligned}&\geq \frac{1}{|S|}\sum_{x\in S} q_x + \sum_{x\in S_{\text{high}}}(p_x - 1/|S|)q + \sum_{x\in S_{\text{low}}}(p_x - 1/|S|)q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|} + \sum_{x\in S} p_x \cdot q - \sum_{x\in S}\frac{1}{|S|}\cdot q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|} + q - q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|}.\end{aligned}$$

The last quantity $\frac{\mathbb{E}[c_i]\cdot n}{|S|}$ is precisely the expected hit rate under the uniform distribution, as we argued in Section 4. Therefore, the expected hit rate under any non-uniform distribution is always lower bounded by the hit rate under the uniform distribution.

D Analysis on Warm-up Time

In the following, we present an informal estimation on the relationship between the hit rate of BLP and its warm-up time. As we argued in Section 4.5, the hit rate is equal to the number of occupied entries in the BLP divided by the size of the working set. In each lookup in the warm-up period, the key may be either a) in the BLP already, or b) not in the BLP and the buckets are full, or c) not in the BLP and the bucket is not full. Only in case c), do we increase the number of occupied entries by one. Denote the *final* hit rate by $r_{\max}$. When the current hit rate is r, we are going to *approximate* the probability of case c) by $r_{\max} - r$. That is, we assume there is a fixed set of $(r_{\max} - r) \cdot (\alpha mn)$ keys in the working set such that they are the missing keys from the BLP in order to achieve the maximum hit rate of $r_{\max}$.

Therefore, let L be the length of the warm-up, we have

$$\frac{\mathrm{d}r}{\mathrm{d}L} = \frac{r_{\max} - r}{\alpha mn},$$

and when $L = 0, r = 0$. By solving this ordinary differential equation, we obtain

$$r = r_{\max}(1 - e^{-\frac{L}{\alpha mn}}).$$

That is, when the length of the warm-up is a large constant times the working set size, the estimated hit rate becomes very close to $r_{\max}$. For $\alpha = 0.95$, we have verified by experiments that a warm-up period of length 20 times the working set size is sufficient to obtain a hit rate that is less than 1% lower than $r_{\max}$.

[7] Note that this is not true if the hash function is fixed.

Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks

Alireza Farshin[*†]
KTH Royal Institute of Technology

Amir Roozbeh[*]
KTH Royal Institute of Technology
Ericsson Research

Gerald Q. Maguire Jr.
KTH Royal Institute of Technology

Dejan Kostić
KTH Royal Institute of Technology

Abstract

Memory access is the major bottleneck in realizing multi-hundred-gigabit networks with commodity hardware, hence it is essential to make good use of cache memory that is a faster, but smaller memory closer to the processor. Our goal is to study the impact of cache management on the performance of I/O intensive applications. Specifically, this paper looks at one of the bottlenecks in packet processing, i.e., direct cache access (DCA). We systematically studied the current implementation of DCA in Intel® processors, particularly Data Direct I/O technology (DDIO), which directly transfers data between I/O devices and the processor's cache. Our empirical study enables system designers/developers to optimize DDIO-enabled systems for I/O intensive applications. We demonstrate that optimizing DDIO could reduce the latency of I/O intensive network functions running at 100 Gbps by up to ~30%. Moreover, we show that DDIO causes a 30% *increase* in tail latencies when processing packets at 200 Gbps, hence it is crucial to selectively inject data into the cache or to explicitly bypass it.

1 Introduction

While the computer architecture community continues to focus on hardware specialization, the networking community tries to achieve greater flexibility with Software-defined Networking (SDN) together with Network Function Virtualization (NFV) by moving from specialized hardware toward commodity hardware. However, greater flexibility comes at the price of lower performance compared to specialized hardware. This approach has become more complex due to the end of Moore's law and Dennard scaling [14]. Furthermore, commercially available 100-Gbps networking interfaces have revealed many challenges for commodity hardware to support packet processing at multi-hundred-gigabit rates. More specifically, the interarrival time of small packets is shrinking to a few nanoseconds (i.e., less than Last Level Cache (LLC) latency). Consequently, any costly computation prevents commodity hardware from processing packets at these rates, thereby causing a tremendous amount of buffering and/or packet loss. As accessing main memory is impossible at these line rates, it is essential to take *greater* advantage of the processor's cache [81]. Processor vendors (e.g., Intel®) introduced new monitoring/controlling capabilities in the processor's cache, e.g., Cache Allocation Technology (CAT) [59]. In alignment with the desire for better cache management, this paper studies the current implementation of Direct Cache Access (DCA) in Intel processors, i.e., Data Direct I/O technology (DDIO), which facilitates the direct communication between the network interface card (NIC) and the processor's cache while avoiding transferring packets to main memory. Our goal is to complete the recent set of studies focusing on understanding the leading technologies for fast networking, i.e., Peripheral Component Interconnect express (PCIe) [58] and Remote Direct Memory Access (RDMA) [37]. We believe that understanding & optimizing DDIO is the missing piece of the puzzle to realize high-performance I/O intensive applications. In this regard, we empirically reverse-engineer DDIO's implementation details, evaluate its effectiveness at 100/200 Gbps, discuss its shortcomings, and propose a set of optimization guidelines to realize performance isolation & achieve better performance for multi-hundred-gigabit rates. Moreover, we exploit a little-discussed feature of Xeon® processors to demonstrate that *fine-tuning* DDIO could improve the performance of I/O intensive applications by up to ~30%. To the best of our knowledge, we are the first to: (i) systematically study and reveal details of DDIO and (ii) take advantage of this knowledge to process packets more efficiently at 200 Gbps.

Why DCA matters? Meeting strict Service Level Objectives (SLO) and offering bounded latency for Internet services is becoming one of the critical challenges of data centers while operating on commodity hardware [54]. Consequently, it is essential to identify the sources of performance variability in commodity hardware and *tame* them [51]. In computer

[*] Both authors contributed equally to the paper.
[†] This author has made all open-source contributions.

systems, one of these sources of variability is the cache hierarchy, which can introduce uncertainty in service times, especially in tail latencies. Additionally, the advent of modern network equipment [82] enables applications to push costly calculations closer to the network while keeping & performing only stateful functions at the processors [36, 38], thereby making modern network applications ever *more* I/O intensive. Hence, taming the performance variability imposed by the cache, especially for I/O, is now more crucial than before. Moreover, as CPU core count goes up, it is important to be able to deliver appropriate I/O bandwidth to them. Therefore, we *go one level deeper* [61] to investigate the impact of I/O cache management, done by DCA, on the performance of multi-hundred-gigabit networks.

Contributions. In this paper, we:

① Design a set of micro-benchmarks to reveal little-known details of DDIO's implementation[*] (§4),

② Extensively study the characteristics of DDIO in different scenarios and identify its shortcomings[*] (§5),

③ Show the importance of balancing load among cores and tuning DDIO capacity when scaling up (§6),

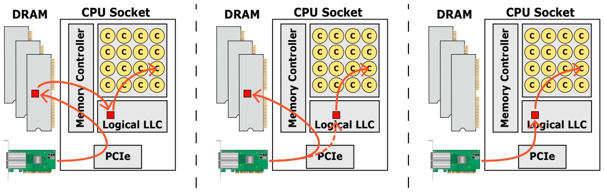
④ Measure the sensitivity of multiple applications (i.e., Memcached, NVMe benchmarks, NFV service chains) to DDIO (§7),

⑤ Demonstrate the necessity and benefits of bypassing cache while receiving packets at 200 Gbps (§8),

⑥ Discuss the lessons learned from our study that are essential for optimizing DDIO-enabled systems receiving traffic at multi-hundred-gigabit rates (§9).

2 Direct Cache Access (DCA)

A standard method to transfer data from an I/O device to a processor is Direct Memory Access (DMA). In this mechanism, a processor, typically instructed by software, provides a set of memory addresses, aka receive (RX) descriptors, to the I/O device. Later, the I/O device directly reads/writes data from/to main memory without involving the processor. For inbound traffic, the processor can be informed about newly DMA-ed data either by receiving an interrupt or polling the I/O device. Next, the processor fetches the I/O data from main memory to its cache in order to process the data. For outbound traffic, the processor informs the I/O device (via transmit (TX) descriptors) of data that is ready to be DMA-ed from main memory to the device. The main source or destination of traditional DMA transfers is main memory, see Fig. 1a. However, the data actually needs to be loaded into the processor's cache for processing. Therefore, this method is inefficient and costly in terms of (i) number of accesses to main memory [43] (i.e., $2n+5$ for n cache lines [43]), (ii) access latency to the I/O data, and (iii) memory bandwidth usage. Moreover, the negative impact of these inefficiencies becomes increasingly severe with higher link speeds. For instance, a server has 6.72 ns to process small packets at 100 Gbps, whereas every access to main memory takes ~100 ns, 15× more expensive. Therefore, placing the I/O data directly in the processor's cache rather than in main memory is desirable. The advent of faster I/O technologies motivated researchers to introduce Direct Cache Access (DCA) [25, 42, 43]. DCA exploits PCIe Transaction Layer Packet Processing Hint [30], making it possible to prefetch portions of I/O data to the processor's cache, see Fig. 1b. Potentially, this overcomes the drawbacks of traditional DMA, thereby achieving maximal I/O bandwidth and reducing processor stall time. Although this way of realizing DCA can effectively prefetch the desired portions of I/O data (e.g., descriptors and packet header), it is still inefficient in terms of memory bandwidth usage since the whole packet is DMA-ed into main memory. Additionally, this requires operating system (OS) intervention and support from the I/O device, system chipset, *and* processor [1]. To address these limitations and avoid ping-ponging data between main memory & the processor's cache, Intel rearchitected the prefetch hint-based DCA, introducing Data Direct I/O technology (DDIO) [28].



(a) Trad. DMA. (b) DCA. (c) DDIO.

Figure 1: Different approaches of DMA for transferring data from an I/O device (e.g., NIC). Red arrows show the path that a packet traverses *before* reaching the processing core.

3 Data Direct I/O Technology (DDIO)

Intel introduced DDIO technology with the Xeon E5 family. With DDIO, I/O devices perform DMA directly to/from Last Level Cache (LLC) rather than system memory, see Fig. 1c. DDIO is also known as write-allocate-write-update-capable DCA (wauDCA) [45], as it uses this policy to update cache lines in an n-way set associative LLC, where n cache lines form one set. For packet processing applications, NICs can send/receive both RX/TX descriptors and the packets themselves via the LLC, thereby improving applications' response time & throughput[†]. DDIO works as follows [41]:

Writing packets. When a NIC writes a cache line to LLC via PCIe, DDIO overwrites the cache line if it is already present in *any* LLC way (aka a PCIe write hit or *write update*). Otherwise, the cache line is allocated in the LLC and DDIO writes the data into the newly allocated cache line (aka a PCIe write miss or *write allocate*). In the latter case, DDIO is restricted to use only a limited portion of LLC when allocating

[*] The source code is available at: https://github.com/aliireza/ddio-bench

[†] We will use the terms I/O device and NIC interchangeably.

cache lines. It is possible to *artificially* increase this portion by warming up the cache with processor writes to the address of these buffers, then DDIO performs write-updates [16].

Reading packets. A NIC can read a cache line from LLC if the cache line is present in *any* LLC way (aka a PCIe read hit). Otherwise, the NIC reads a cache-line-sized chunk from system memory (aka a PCIe read miss).

To monitor DDIO and its interaction with I/O devices, Intel added uncore performance counters to its processors [29]. The Intel Performance Counter Monitor (PCM) tool (e.g., `pcm-pcie.x`[*]) [86] can count the number of PCIe write hits/misses (represented as an ItoM event) and PCIe read hits/misses (represented as a PCIeRdCur event). Next, we discuss the inherent problem of DDIO, which makes it hard to achieve low-latency for multi-hundred-gigabit NICs.

3.1 How can DDIO become a Bottleneck?

Researchers have shown some scenarios in which DDIO cannot provide the expected benefits [11, 41, 50, 83]. Two typical cases occur when new incoming packets repeatedly evict the previously DMA-ed packets (i.e., *not-yet-processed* and *already-processed packets*) in the LLC. Consequently, the processor has to load not-yet-processed packets from main memory rather than LLC and the NIC needs to DMA the already-processed packets from the main memory, thereby missing the benefits of DDIO. Tootoonchian et al. referred to this problem as *the leaky DMA problem* [83]. To mitigate this problem, they proposed reducing the number of "in-flight" buffers (i.e., descriptors) such that all incoming packets fit in the limited portion of LLC used for I/O. Thus, performance isolation can be done using *only* CAT (i.e., cache partitioning). Unfortunately, reducing the number of RX descriptors is only a temporary solution due to increasing link speeds. Multi-hundred-gigabit NICs introduce new challenges, specifically:

① **Packet loss.** At sub-hundred-gigabit link speeds reducing the number of RX descriptors may not result in a high packet loss rate, but at $\geq$100 Gbps packet loss increases due to the tight processing time budget before buffering/queuing happens. For instance, every extra ~7 ns spent stalling or processing/accessing a packet causes another packet to be buffered when receiving 64-B packets at 100 Gbps. When there are insufficient resources for immediate processing, increasing the number of RX descriptors permits packets to be buffered rather than dropped. Delays in processing might occur because of interrupt handling, prolonged processing, or a sudden increase in the packet arrival rate [17]; therefore, multi-hundred-gigabit networks cannot avoid packet loss without having a *sufficiently large* number of descriptors. Increasing the number of processing cores can reduce the packet loss rate, but applications that are compute- or memory-intensive require many cores to operate at the speed of the underlying hardware, e.g., Thomas et al. [81] mention that a server performing *one* DRAM access per packet needs 79 cores to process packets at 400 Gbps.

② **TX buffering.** One of the scenarios that makes DDIO inefficient is the eviction of already-processed packets. Reducing the number of RX descriptors may solve this problem for systems that require a small number of TX descriptors, but this is not the case for 100-Gbps NICs. Unfortunately, the *de facto* medium for DMA-ing packets (i.e., PCIe 3.0) induces some transmission limitations [58]. Consequently, packets often need to be buffered in the computer system for some time before being DMA-ed to the NIC. This buffering can be realized by either a software queue or increasing the number of TX descriptors [35]. Unfortunately, either of these alternatives increases the probability of eviction of already-processed packets. Therefore, completely solving the leaky DMA problem requires fine-tuning *both* the size of the software queue and the number of RX & TX descriptors.

③ **PAUSE frames.** To alleviate packet loss, one can use Ethernet flow control mechanisms (e.g., PAUSE frames) that cause packets to be buffered earlier in the network, i.e., PAUSE frames stop the previous network node from transmitting packets for a short period. However, these mechanisms are costly in terms of latency, making them less desirable than packet loss for time-critical applications. The minimum and maximum pause duration of a 100-Gbps interface are 5.12 ns and 335.5 µs [56]. Our measurements show that a core that is simply forwarding packets at 100 Gbps with 1024 RX & TX descriptors causes the NIC to send ~179 k PAUSE frames while receiving ~80 M packets.

Dynamic reduction. As reducing the number of RX buffers cannot *fully* solve the problem and it shifts the problem to another part of the network, most probably the previous node; therefore, an alternative is to *dynamically* reduce the pressure on the LLC when the number of I/O caused cache evictions starts to increase[†]. These cache evictions can be tracked by monitoring either PCIe events or the length of the software queue. After detecting a problem, the processor should fetch a smaller number of packets from the NIC (i.e., reducing the RX burst size). Thus, the processor passes fewer free buffers to the NIC, reducing the number of DMA transactions. Unfortunately, this approach does not perform well, hence we need a *proactive* solution, not a reactive one.

Is it sufficient to scale up? Due to the demise of the Dennard scaling [14], processors are now shipped with more cores rather than higher clock frequencies. Moreover, the per-core cache quota (i.e., LLC slices) has decreased in recent Xeon processors, i.e., the size of LLC slices reduced from 2.5 MiB to 1.375 MiB in the Xeon scalable family (i.e., Skylake) [55]. This reduction in per-core cache size directly affects the optimal number of descriptors as these are proportional to the limited space for DDIO. For instance, using 18 cores, each having 256 RX descriptors, requires ~6.5 MiB, which is equal

[*] The description of events can be found in [27] and pp. 63-66 of [41].

[†] Our implementation is available at: https://github.com/tbarbette/fastclick/tree/DMAdynamic

to ~26.6% of the LLC in this processor and greater than the available DDIO capacity (see §4.1).

Our approach. To overcome these challenges, it is necessary to study and analyze DDIO empirically in order to *make the best use of it*. A better understanding of DDIO and its implementation can help us optimize current computer systems and enables us to propose a better DCA design for future computer systems that could accommodate the ever-increasing NIC link speeds. For instance, Fig. 2 demonstrates that tuning DDIO's capacity makes it possible to achieve a suitable performance while using a large number of descriptors (our approach), as opposed to using a limited number of descriptors (ResQ's approach proposed by Tootoonchian et al. [83]).

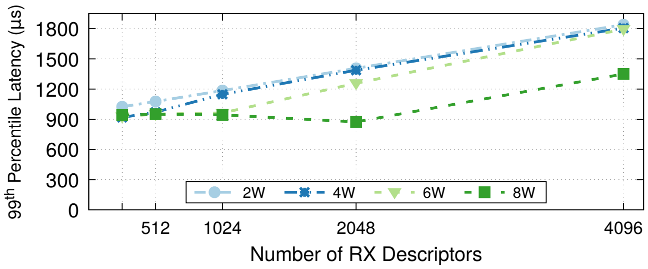


Figure 2: Using more DDIO ways ("W") enables 2 cores to forward 1500-B packets at 100 Gbps with a larger number of descriptors while achieving better or similar tail latency.

4 Understanding Details of DDIO

This section discusses four questions: ① What part of LLC is used for I/O? ② How does I/O interact with other applications? ③ Does DMA via remote sockets pollute LLC? and ④ Is it possible to disable/tune DDIO?

Testbed. We use a testbed with the configuration shown in Table 1 running Ubuntu 18.04.2 (Linux kernel-4.15.0-54). We use the Skylake server unless stated otherwise. FastClick [9] is used to generate & process packets. Additionally, we use a campus trace as a real workload (with mixed-size packets) and generate synthetic traces (with fixed-size packets). For our multicore experiment, we use RSS [24] to distribute packets among different queues (one queue per core), unless stated otherwise. Furthermore, we isolate the one CPU socket on which we run the experiment to increase the accuracy of the measurements. PAUSE frames are disabled to avoid taking into account pause duration in the end-to-end latency. In all experiments, the NIC driver sets the appropriate number of TX descriptors based on the number of TX queues, and to avoid extra looping at the transmitting side FastClick buffers up to 1024 packets. We use the Network Performance Framework (NPF) tool [57] to run the experiments.

4.1 Occupancy

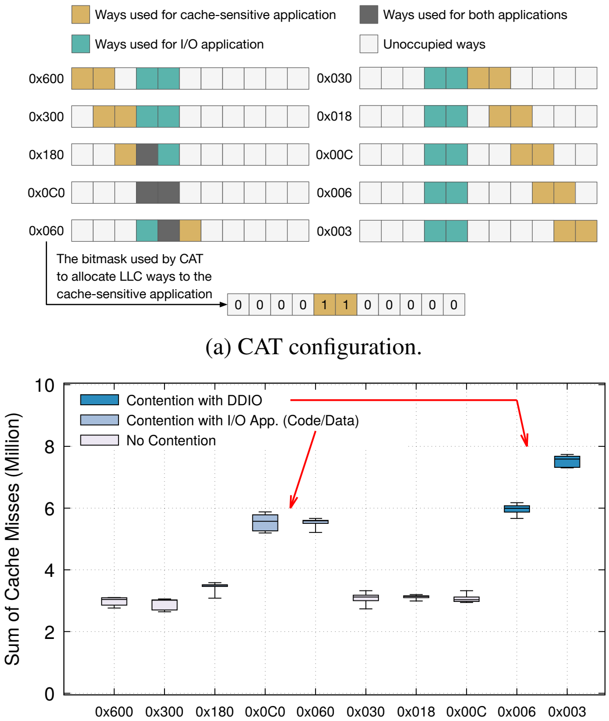
Initially, Intel announced that DDIO only uses 10% of LLC [28] and did not mention what *part* of the LLC is used (i.e., ways, sets, or slices [15]). Recent Intel technical reports mention that DDIO only uses a subset of LLC ways, by default two ways [41, 72]. However, it is still unclear whether this "subset" is fixed or whether it can be dynamically selected using a variant of Least Recently Used (LRU) policies [33, 34, 65, 87]. Knowledge of these details could avoid I/O contention and optimize performance isolation [83] by performing precise cache management/partitioning [13, 62] (e.g., way partitioning with CAT [59]). This issue becomes increasingly critical for newer generations of Xeon processors that have lower LLC set-associativity (e.g., 11 ways in some Skylake processors, as opposed to 20 ways in Haswell processors), thereby using a larger portion ($\frac{2}{11} \approx 18\%$) of the LLC for I/O. Lower set-associativity makes the cache less flexible when the LLC is divided into multiple partitions, each of which could be used to accommodate different applications' code & data. To clarify this, we assumed that the ways that are used for DDIO are *fixed* and then try to confirm this with an experiment in which we co-run an I/O and a cache-sensitive application. To increase the pressure on the LLC by DMA-ing more cache lines, we used an L2 forwarding DPDK-based application as the I/O intensive application. Specifically, it receives large packets (1024-B) at a high rate (~82 Gbps) using a large number of RX descriptors (4096 RX descriptors). For the cache-sensitive application, we chose `water_nsquared` from the Splash-3 benchmark suite [62, 66, 69] since it performs a large number of LLC accesses; hence, it interferes with the I/O application.

Each application is run on a different core and CAT is used to allocate different cache ways to each core. We allocate two *fixed* ways to the I/O application and two *variable* ways to the cache-sensitive application. To avoid memory bandwidth contention, we also used Memory Bandwidth Allocation (MBA) technology [21] to limit the memory bandwidth of each core to 40%. Fig. 3a shows the CAT configuration used in the experiment. We start by allocating the two leftmost ways (i.e., bitmask of 0x600) to the cache-sensitive application and then we keep shifting the allocated ways one

Table 1: Details of our testbed. In each case, the NIC is a Mellanox ConnectX-5 VPI.

Configuration / Machine	Intel Xeon Processor			Memory	Last Level Cache (LLC)	
	Model	Frequency	#Cores		Size	Associativity
Packet generator (Skylake)	Gold 6134	3.2 GHz	8	512 GiB	18×1.375 MiB	11
Server (Skylake)	Gold 6140	2.3 GHz	18	256 GiB	18×1.375 MiB	11
Server (Haswell)	E5-2667 v3	3.2 GHz	8	128 GiB	8×2.5 MiB	20

to the right until we cover all the LLC ways while measuring the LLC misses of the I/O application. Fig. 3b shows the results of this experiment. These results demonstrate that the cache-sensitive application interferes with the I/O application in *two* regions. The first (see 0x0C0 in Fig. 3b) occurs when the cache-sensitive application uses the same ways as the I/O application, due to the code/data interference of the two applications. However, the second (see 0x003 in Fig. 3b) cannot be explained with this same argument since the I/O application is limited to using other ways (i.e., 0x0C0). Furthermore, since the CPU socket is isolated, no other application can cause cache misses. CAT only mitigates the contention induced by code/data not DDIO. Therefore, we conclude that the second interference is most probably due to I/O, which means DDIO uses the two rightmost ways in LLC (i.e., bitmask of 0x003). The interference is proportional to the number of received packets per second × average packet size. We expected to see roughly the *same* amount of cache misses for bitmasks of 0x180 and 0x060, as they are completely symmetrical in terms of way occupancy. However, the undocumented LRU policy of the CPU may affect how the application uses the cache ways.



(a) CAT configuration.

(b) Sum of cache misses for the I/O application.

Figure 3: Interference of an I/O and a cache-sensitive application using the `parsec_native` configuration (to cause a high rate of cache misses) when the cache-sensitive application uses different LLC ways. The rise in the rightmost side shows the contention with DDIO ways.

4.2 I/O Contention

One of the established mechanisms to ensure performance isolation and mitigate cache contention is CAT, which limits different applications to a subset of LLC ways. However, §4.1 showed that DDIO uses two *fixed* LLC ways. Therefore, isolating applications using CAT may not *fully* ensure performance isolation, due to cache contention caused by I/O. Such contention may occur in two common scenarios:

① **I/O vs. Code/Data.** When an application is limited to using those ways which are also used by DDIO, then cache lines allocated in LLC for DDIO may evict the code/data of any application (i.e., either I/O or non-I/O application). This issue was discussed by Tootoonchian et al. [83]. Their proposed framework, ResQ, uses only 90% of LLC to avoid interfering with DDIO's reserved space, but does not mention which part of LLC is isolated. §4.1 showed the destructive (i.e., ~2.5×) impact on the cache misses of the I/O application due to a cache-hungry application overlapping with DDIO, see the rise in cache misses at the right side of Fig. 3b. However, it did not show the impact of contention on the cache-hungry application; therefore, we repeated the experiment and measured the cache misses of the cache-sensitive application while using a lighter configuration. Fig. 4 illustrates that the cache misses of the cache-sensitive application were similarly adversely affected. Therefore, overlapping any application with DDIO ways in LLC can reduce the performance of *both* applications. To tackle this, one can isolate the I/O portion of LLC (e.g., the two ways used for DDIO) by using CAT so that applications share the LLC *without* overlapping with I/O. Comparing Fig. 3b and 4, we see that an unexpected rise (almost 3×) in cache misses occurs in a different region (i.e., bitmask of 0x600 in Fig. 4 as opposed to bitmask of 0x003 in Fig. 3b) when I/O is evicting code/data. Hence, we speculate that CAT does not use a *bijective* function to map I/O & code/data to ways, thus $\boxed{f : \text{code/data} \rightarrow \text{Ways}}$ is not equivalent to $\boxed{g : \text{I/O} \rightarrow \text{Ways}}$. Specifically, I/O evicts code/data when the latter is located in the two leftmost ways whereas code/data evicts I/O when the latter is using the two rightmost ways. Such information is useful to know, as it will give us an understanding of the eviction policy and the default priority of code/data and I/O.

② **I/O vs. I/O.** When multiple I/O applications are isolated from each other with CAT, they could still *unintentionally* compete for the *fixed* ways allocated to DDIO. §8.1 elaborates the negative impact of this type of contention.

Security implication. Since DDIO uses two *fixed* ways in LLC, it is possible to extend microarchitectural attacks to extract useful information from I/O data (e.g., NetCAT [44] and Packet Chasing [76, 77]). Furthermore, I/O applications can be vulnerable to performance attacks.

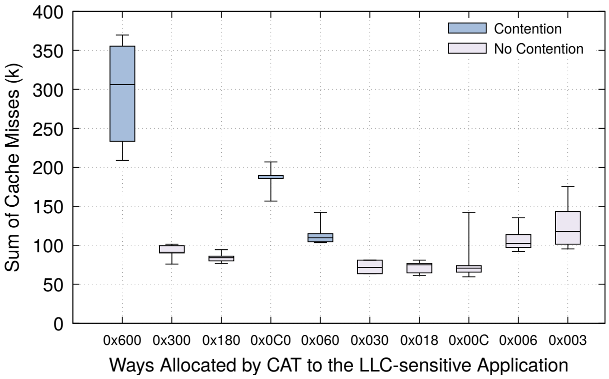


Figure 4: Interference of the cache-sensitive and the I/O applications. Y axis shows the sum of cache misses of the cache-sensitive application. The cache-sensitive application uses a lighter configuration (i.e., `ddio_sim`), which causes fewer cache misses than the I/O application.

4.3 DMA via Remote Socket

According to Intel [16, 32], the current implementation of DDIO only affects the local socket. Consequently, if a core accesses I/O data from an I/O device connected to a remote socket, the data has to traverse the inter-core interconnect, i.e., Intel QuickPath Interconnect (QPI) or Intel Ultra path Interconnect (UPI). It was uncertain whether data traversing the inter-core interconnect is loaded into the LLC of the remote socket or not. We clarified this by running the same experiment discussed in §4.2 while the NIC is connected to a remote socket. The result (removed for brevity) showed that cache misses of neither application were affected by the *I/O* cache lines, hence packets coming through the UPI links *do not end up in the local LLC*. Additionally, the cache misses of the I/O application dramatically increased to 20× greater than when receiving packets via the local socket without any contention. Thus, DDIO is ineffective for the remote socket *and* it pollutes the LLC on the socket connected to the NIC.

4.4 Tuning Occupancy and Disabling DDIO

Although [20, 72] mention that DDIO uses two ways by *default*, there is no mention of whether it is *possible* to increase or decrease the number of ways used by DDIO. A little-discussed Model Specific Register (MSR) called "IIO LLC WAYS" with the address of 0xC8B[*] is discussed in a few online resources [64, 79] and server manuals [73, 74]. For Skylake, the default value of this register is equal to 0x600 (i.e., two bits set). While these bits cannot be unset, it is possible to set additional bits and the maximum value for this register on our CPU is 0x7FF (i.e., 11 bits set: the same as the number of LLC ways). New values for this register follow the same format as CAT bitmasks. On a processor with the Skylake microarchitecture, these new values should contain consecutive ones, while the Haswell microarchitecture does not require this (i.e., allowing any value in [0x60000, 0xFFFFF]).

To see whether this MSR register has an effect on performance, we measured the PCIe read/write hit rates (i.e., ItoM and PCIeRdCur events) while using different values for IIO LLC WAYS. We calculate the hit rate based on the number of hits and misses during an experiment where an I/O application processes packets of 1024 B at 100 Gbps while using 4096 RX descriptors. Fig. 5 shows that increasing the value of this MSR register leads to a higher PCIe read/write hit rate. This suggests that increasing the value of this register could improve the ability of the system to handle packets at high rates. We believe that the value of this register is *positively* correlated with the fraction of LLC used by DDIO. Using the technique in §4.1, we could not detect the newly added I/O ways, thus we speculate that the newly added ways follow a different policy (e.g., LRU) than the first two ways used for I/O. Therefore, we assume that the number of bits set specifies the number of ways used by DDIO.

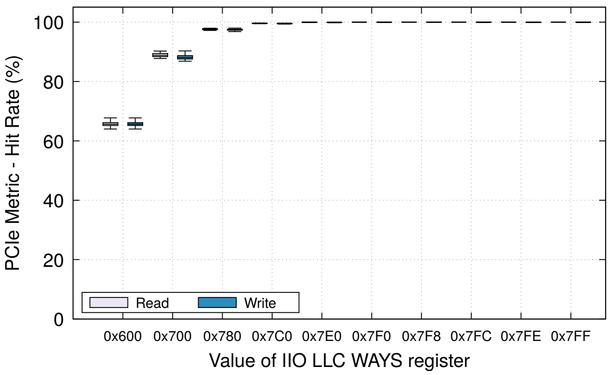


Figure 5: Tuning IIO LLC WAYS register increases PCIe read/write hit rates. The achieved throughput is 82-86 Gbps in this experiment.

Disabling DDIO. DDIO is bundled as a part of Intel Virtualization Technology (Intel VT), hence it is possible to enable/disable it in BIOS for some vendors [16, 23, 88]. According to [44, 72], DDIO can be disabled globally (i.e., by setting the `Disable_All_Allocating_Flows` bit in "iiomiscctrl" register) **or** per-root PCIe port (i.e., setting bit `NoSnoopOpWrEn` and unsetting bit `Use_Allocating_Flow_Wr` in "perfctrlsts_0" register). Some brief discussions of the benefits of disabling DDIO exist [11, 78], but we elaborate this more thoroughly in §7. We implemented an element for FastClick, called *DDIOTune*, which can enable/disable/tune DDIO[†].

[*] One can read/write this register via `msr-tools` (e.g., rdmsr and wrmsr).

[†] The element is available at: https://github.com/tbarbette/fastclick/wiki/DDIOTune

5 Characterization of DDIO

This section scrutinizes the performance of DDIO in different scenarios while exploiting the tuning capability of DDIO. The goal is to show where DDIO becomes a bottleneck and when tuning DDIO matters. Therefore, we examined the impact of both system parameters (i.e., #RX descriptors, #cores, and processing time) and workload characteristics (i.e., packet size and rate) on DDIO performance. All of these measurements were done 20 times for both Skylake and Haswell microarchitectures. We observed the same behavior in both cases, but only discuss the Skylake results for the sake of brevity. We initially focus on the performance of an L2 forwarding network function, as an example of an I/O intensive application. Later, we discuss the impact of applications requiring more processing time per packet.

5.1 Packet Size and RX Descriptors

§3.1 discussed the negative consequence of a large number of RX descriptors on DDIO performance. This section continues this discussion by looking at the PCIe read/write hit rate metrics for different numbers of RX descriptors and different packet sizes. Fig. 6 shows the results of our experiments for PCIe write hit rate. PCIe read hit rates (not included for brevity) demonstrate similar behavior. When packets are >512 B, the PCIe read/write hit rates monotonically decrease with an increasing number of RX descriptors. More specifically, sending 1500-B packets, even with a relatively small number of RX descriptors (i.e., 128), causes 10% misses for both PCIe read and PCIe write hit rates. Furthermore, increasing the number of RX descriptors to 4096 makes DDIO operate at ~40% hit rate, hence 60% of packets require cache allocation and they had to be DMA-ed back to the NIC from main memory rather than LLC. Note that the packet generator is generating packets as fast as possible. Therefore, small packets show the case when the arrival *rate* is maximal, while large packets demonstrate maximal *throughput*, see Fig. 7.

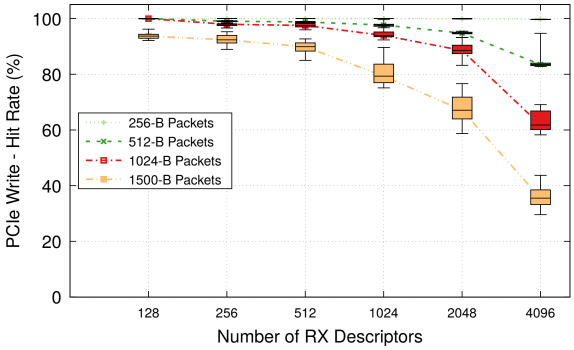
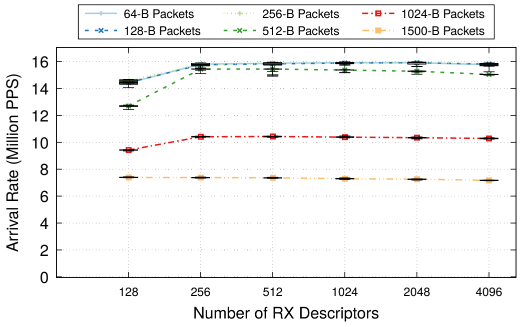
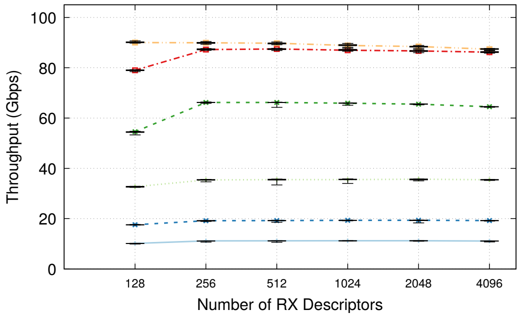


Figure 6: Increasing the number of descriptors and/or packet size *adversely* affects the performance of 2-way DDIO, while one core is forwarding packets at the maximum possible rate. We removed the results for 64-B and 128-B packets, as they show a behavior similar to 256-B packets.



(a) Arrival rate.



(b) Throughput.

Figure 7: Increasing the packet size reduces the arrival rate, i.e., the number of received/processed packets per second, due to NIC and PCIe limitations. Note that our testbed cannot exceed 90 Gbps when *only* one core is forwarding packets.

Unexpected I/O evictions. In some cases (e.g., 1500-B packets with 128 RX descriptors in Fig. 6), the size of the injected data is smaller than the DDIO capacity (i.e., 187.5 KiB $\ll$ 4.5 MiB). Even taking into account the TX descriptors and the FastClick's software queue, the maximum cache footprint of this workload is ~2 MiB. However, DDIO still experiences ~10% misses. We believe that this behavior may occur when an application cannot use the whole DDIO capacity due to (i) the undocumented cache replacement policy and/or (ii) the cache's complex addressing [15], thus multiple buffers may be loaded into the same cache set.

5.2 Packet Rate and Processing Time

§5.1 demonstrated that DDIO performs extremely poorly when a core does minimal processing at 100 Gbps. Next, we focus on the worst-case scenario of the previous experiment (i.e., sending 1500-B packets with 4096 RX descriptors) while changing the packet rate. To achieve 100 Gbps, we use two cores. Fig. 8 shows the PCIe read and PCIe write hit rates. The PCIe read metric results reveal that DDIO performs relatively well until reaching 98 Gbps. However, the PCIe write results indicate that DDIO has to continually allocate cache lines in LLC for 25% of packets at most of these throughputs, due to insufficient space for all of the buffers. Moreover, throughputs above 75 Gbps exacerbate this problem.

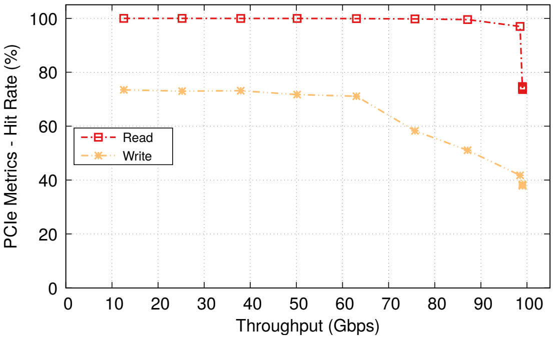


Figure 8: Increasing packet rates *negatively* impact the PCIe metrics, when 2 cores forward 1500-B packets with 4096 RX descriptors. The PCIe write metric is more degradation-prone.

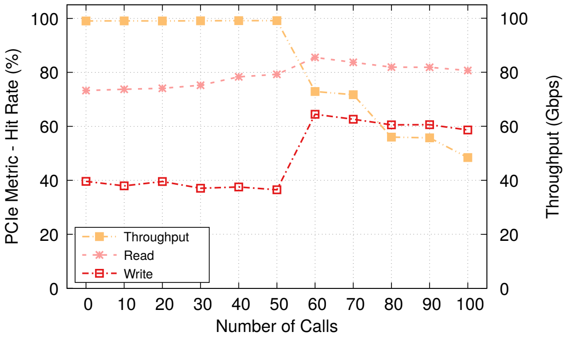


Figure 9: Making an application more compute-intensive results in better PCIe metrics, but *lower* throughput. In addition to forwarding packets, two cores call a dummy computation, while receiving 1500-B packets with a total of 4096 RX descriptors at 100 Gbps.

So far, we analyzed DDIO performance when cores performed minimal processing (i.e., swapping MAC addresses). Now, we analyze DDIO performance for more compute/memory-intensive I/O applications. Memory-intensive applications access memory frequently and execute few instructions per memory access. The time to accessing memory differs depending upon the availability of a cache line in a given part of the memory hierarchy. Therefore, we focus on the number of CPU cycles of the computation; noting that a memory access can be accounted for as given number of cycles. Note that increasing the processing time can change the memory access pattern, as packets continue to be injected by the NIC while some packets are enqueued in the LLC. To see the impact of different packet processing times on the performance of DDIO, we vary the amount of computation per packet by calling the `std::mt1993` random number generator multiple times. Ten such calls take ~70 cycles. Fig. 9 illustrates the effect of increasing per-packet processing time on the PCIe metrics & achieved throughput. These results demonstrate that increasing processing time slightly improves PCIe read hits rates up to ~60 calls, i.e., 400 cycles. This is expected, as increasing processing makes the application less I/O intensive as the application provides buffers to the NIC at a slower pace. However, increasing processing causes the available processing power (i.e., #cores) to become a bottleneck, substantially decreasing throughput. Similarly, PCIe write hit rates increases after exceeding 60 calls, due to a decrease in throughput & amount of cache injection. Therefore, DDIO performance *matters most* when an application is I/O bound, rather than CPU/memory bound.

5.3 Numbers of Cores and DDIO Capacity

When processing power limits an application's performance, the system should scale up/out. However, this scaling can affect DDIO's performance. To see the effect of scaling up, we measured the PCIe metrics while different numbers of cores were forwarding large packets. Fig. 10 shows that when an application is I/O intensive, increasing the number of cores improves the PCIe read/write hit rate, as it enhances the packet transmission rate because of more TX queues and faster consumption of packets enqueued in the LLC. To avoid synchronization problems, every queue is bound to one core. However, beyond a certain point (i.e., four cores in our testbed), increasing the number of cores causes *more* contention in the cache, as every core loads packets independently into the limited DDIO capacity. Furthermore, since newer processors are shipped with more cores, scaling up, even with a small number of RX descriptors, eventually causes the leaky DMA problem–the same problem as having a large number of descriptors (see §3.1).

Fig. 11 shows PCIe metrics for 1, 2, and 4 cores while changing the number of DDIO ways. Comparing the DDIO performance of different numbers of cores/DDIO ways, we conclude that increasing DDIO capacity leads to similar improvements for PCIe metrics. Therefore, increasing the DDIO capacity rather than the number of cores is beneficial when an application's bottleneck is *not* processing power or number of TX queues. Unless scaling up happens efficiently, some cores may receive more packets than others, causing

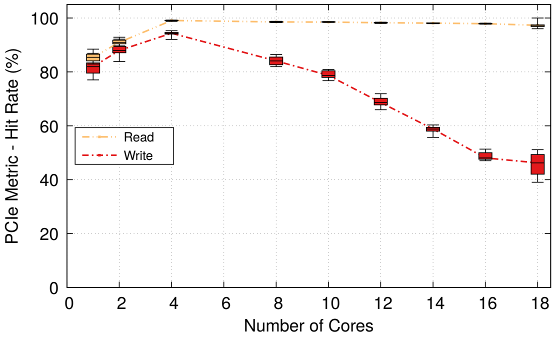


Figure 10: Increasing the number of cores does *not* always improve PCIe metrics for an I/O intensive application. Different numbers of cores are forwarding 1500-B packets at 100 Gbps with 256 RX descriptors per core.

performance degradation. We discuss the impact of load imbalance on DDIO performance in the next section.

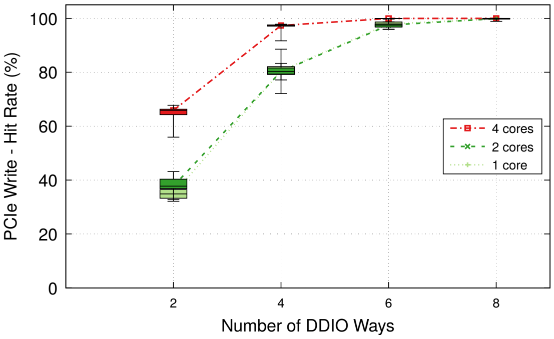


Figure 11: Increasing the number of DDIO ways can have a similar *positive* effect as increasing the number of processing cores, while forwarding 1500-B packets at 100 Gbps with a total of 4096 RX descriptors. PCIe read hit rate shows the same behavior as PCIe writes.

6 Application-level Performance Metrics

The previous section focused on the PCIe read/write hit rates and showed that increasing link speed & packet size and the number of descriptors & cores could degrade these metrics. PCIe read/write hit rates represent the percentage of I/O evictions (i.e., the performance of DDIO), but also *indirectly* affect application performance. The correlation between PCIe metrics and meaningful performance metrics (e.g., latency and throughput) depends on an application's characteristics. For instance, a low PCIe write hit rate can severely affect an application that requires the whole DMA-ed data. Conversely, the impact is much less for an application that needs only a subset of the DMA-ed packet. Fig. 2 showed one example of this correlation for the latter case, where the application only accessed the packet header. These results showed that *even* when an application does not require the whole DMA-ed data, increasing the number of descriptors (i.e., causing a reduction in PCIe hit rate metrics) could negatively affect the 99th percentile latency. Note that we observed the same effect at median latency. This section further elaborates this impact in two scenarios where a stateful network function is processing a realistic workload[*] via 18 cores with a run-to-completion model [38, 93]. The benefits of increasing cache performance are not limited to this model and could be even greater for a pipeline model where fewer cores handle the I/O.

Stateful service chain. To evaluate the effect of increasing DDIO capacity, we chose a stateful service chain composed of a router, a network address port translator (NAPT), and a round-robin load balancer (LB) as a suitable chain to exploit hardware offloading capabilities of modern NICs while still keeping state at the processor. In this case, we offload the routing table of the router to the NIC and only handle the *stateful tasks* (i.e., NAPT + LB) and the basic functionality of the router in software. We generated 2423 IP filter rules for the campus trace using the GenerateIPFlowDirector element in Metron [38] and use DPDK's Flow API technology [31] to offload them into a Mellanox NIC. To examine the impact of load imbalance, we generate two different sets of rules with different load imbalance factors. One distributes the rules among 18 cores in a round-robin manner while the other is load-aware and tries to reduce the flow imbalance in terms of bytes received by every core. We calculated the number of packets received by each core for both cases and the maximum imbalance ratio of a core is 2.78× for the load-aware technique, while the round-robin technique causes 1.69× maximum load imbalance. The load-aware method has a higher load imbalance because we generate rules for the whole trace, but only replay a subset of it. Fig. 12 shows the 99th percentile latency of this chain for different load balancing methods (with different load imbalance ratio), specifically increasing DDIO capacity reduces the 99th percentile latency by ~21% when the load imbalance is higher. However, when the load imbalance is lower, these improvements reduce to ~2%. A higher load imbalance factor means that a core receives more packets than others, some of which could be evicted while enqueued in the LLC. Hence, it is crucial to realize a good balance to get the most out of DDIO. Furthermore, load imbalance is the root cause of many other performance degradations and is hard to prevent [8, 10].

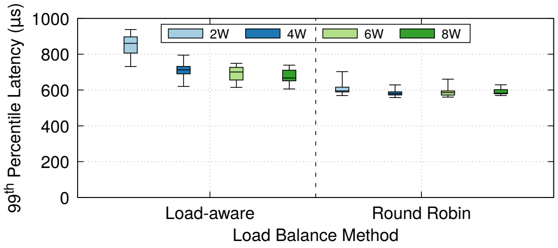


Figure 12: DDIO should be *carefully* tuned when the load imbalance factor is higher. The results shows 99th percentile latency of a stateful network function while 18 cores are processing mixed-size packets at 100 Gbps. The throughputs were 94 & 97 Gbps for load-ware (higher imbalance) & round-robin (lower imbalance) experiments, respectively.

7 Is DDIO Always Beneficial?

The previous section showed that performance could be improved by tuning DDIO for I/O intensive network functions operating at ~100 Gbps. However, these results *cannot* be generalized, as the improvements are highly dependent on the application's characteristics. Moreover, there may be some applications that do not benefit from DDIO tuning. To investigate this, we measure the sensitivity of different applications to DDIO by enabling/disabling it (see §4.4). Table 2 shows the results for four applications/benchmarks:

[*] We replay the first 400 k packets of a 28-minute campus trace fifty times. The full trace has ~800 M packets with an average size of 981 B.

Table 2: DDIO sensitivity changes for different applications.

DDIO / Application	Enabled				Disabled				Sensitivity
	Throughput	Median (μs)	Avg (μs)	99^{th} (μs)	Throughput	Median (μs)	Avg (μs)	99^{th} (μs)	
Memcached (TCP)	1003058 TPS	N/A	477.62	N/A	994387 TPS	N/A	481.62	N/A	Low
Memcached (UDP)	638763 TPS	N/A	750.12	N/A	631354 TPS	N/A	758.75	N/A	Low
NVMe (Full Write)	4427.2 MiB/s	44879.4	44437.6	46452.4	4434.2 MiB/s	44827	44374.68	46452.4	Low
NVMe (Random Read)	3372.4 MiB/s	582	589.67	765.7	3233.7 MiB/s	601.8	614.46	805.7	High
NVMe (Random Write)	1498.3 MiB/s	1307.8	1324.73	1991.2	1499.9 MiB/s	1309.5	1323.38	1971.4	Low
L2 Forwarding	98.01 Gbps	500.82	662	1055.98	87.02 Gbps	1058.15	862	1229.62	High
Stateful Service Chain (without offloading)	63.92 Gbps	665	657	923	63.25 Gbps	672	666	931	Low
Stateful Service Chain (with round-robin offloading)	97.35 Gbps	499	505	595	87.46 Gbps	531	924	1981	High

(i) DPDK-based implementation of Memcached developed by Seastar [5], (ii) an NVMe benchmarking tool (i.e., fio [4]), (iii) L2 forwarding application, (iv) a stateful service chain, used in §6, which performs IP filtering in software rather than offloading it to the NIC, and (v) the stateful service chain with round-robin offloading used in §6. We define sensitivity as "Low" if the maximum impact on the performance of an application is $\leq 5\%$. For Memcached, we use the method recommended by Seastar [2] with 8 instances of memaslap clients running for 120 s and a Memcached instance with 4 cores. For NVMe benchmarks, we tested a Toshiba NVMe (KXG50PNV1T02) with 4×1024-GB SSDs according to [3], where we report the average of 10 runs. The L2 forwarding application forwards mixed-size packets, while using 4 cores with a total of 4096 RX descriptors. The stateful service chain without offloading uses RSS to distribute packets among 18 cores (to increase the throughput) with 18×256 RX descriptors. The results demonstrate that different applications have different levels of sensitivity to DDIO, which can be exploited by system developers to optimize their system in a multi-tenant environment, where multiple I/O applications co-exist, see §8.1. The most sensitive application is L2 forwarding, which is the most I/O intensive application among these applications and can run at line rate. Some applications (e.g., Memcached) experience less benefit from DDIO, as their performance may be bounded by other bottlenecks. A more detailed sensitivity analysis of different applications remains as our future work.

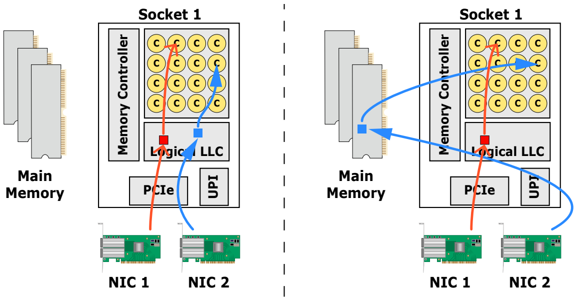
8 Future Directions for DCA

Tuning DDIO occupancy was shown to substantially improve the performance of some applications. However, increasing the portion of the cache used for I/O is only a temporary solution for two reasons: (i) I/O is only a part of packet processing and (ii) to achieve suitable performance many networking applications require a large amount of cache memory for *code/data*. Moreover, many network functions would benefit from performing in-cache flow classification [92]; hence, there is a trade-off between allocating cache to I/O vs. code/data and this trade-off depends on the application's characteristics & cache size. Additionally, since DDIO is way-based, the granularity of partitions is quite coarse in recent Intel processors, due to low set-associativity. Therefore, it is harder to partition the cache fairly between code/data & I/O. These reasons, together with the recent trend in Intel processors of decreasing per-core LLC, eventually make the current implementation of DCA a major bottleneck to achieving low-latency service times. Hence, DCA needs to deliver better performance even with a small fraction of the cache. This makes it necessary to *rethink* the current DCA designs with an eye toward realizing network services running at multi-hundred gigabits per second. Some possible directions/proposals for future DCA are: ① Fine-grained placement: adopting CacheDirector [15] methodology (i.e., sending packets to the appropriate LLC slices) and only sending the relevant parts of these packets to the L2 cache, L1 cache, or *potentially* CPU registers [26]; ② Selective DMA/DCA: only DMA relevant parts of the packet (as required by an application) to the cache and buffer the rest in either main memory, the NIC, or Top-of-Rack switch; and ③ I/O isolation: extend CAT to include I/O prioritization in addition to Code and Data Prioritization (CDP) technology [60] to alleviate I/O contention. These ideas could be simulated in a cycle accurate simulator (e.g., gem5 [6, 12]), which remains as our future work. Next, we examine one potential solution in the current systems to better take advantage of DDIO.

8.1 Bypassing Cache

§3.1 explained that one way to prevent unnecessary memory accesses and the leaky DMA problem is to reduce the number of descriptors. However, this could increase packet loss and generate more PAUSE frames at high link rates. Unfortunately, both can have a severe impact on the service time as they postpone the service time by at least a couple of microseconds. Taking these consequences into account, we believe future DCA technologies should perform cache injection more effectively: *DMA should not be directed to the cache if this would cause I/O evictions*; thus, buffering packets in local memory (at a cost of only several hundreds of nanoseconds) is preferable to dropping or enqueuing packets in previous nodes. Additionally, bypassing cache would be beneficial in a multi-tenant scenario where performance isolation is desired. For instance, low-priority and/or low-DDIO-sensitive applications could bypass cache to make room for high-priority and/or high-DDIO-sensitive applications. In addition, one could prioritize [7] different traffic flows, thus only a subset of received traffic (and hence cores) would use cache for I/O. Implementing a system to prioritize DDIO for different flows either in a programmable switch or modern NICs (e.g., Mellanox Socket Direct Adapters) remains as our future work.

Evaluation. To evaluate the benefits of bypassing the cache, we use two methods: (i) disabling DDIO and (ii) exploiting DMA via a remote socket (see §4.3). We set up a 200-Gbps testbed, see Fig. 13. We first connect two 100-Gbps NICs to the same socket. Next, we connect one of these NICs to a remote socket. We run two instances of L2 forwarding application located on the first socket, each of which uses 4 cores and one NIC to forward mixed-size packets. We chose four cores per NIC because our earlier experiments (see Fig. 10) showed that DDIO can achieve an acceptable performance while receiving 1500-B packets with four cores. To reduce the contention for cache and memory bandwidth, we apply CAT & MBA to each application (similar to ResQ [83]). We assume that one of the applications has a higher priority, and we measure its latency in five different scenarios: **(i)** without the presence of the low-priority application, **(ii)** when the low-priority application pollutes the cache via 2-way DDIO (see Fig. 13a), **(iii)** when the low-priority application pollutes the cache via 4-way DDIO, **(iv)** when the low-priority application bypasses the cache by DMA-ing packets via a remote socket (see Fig. 13b), **(v)** when the low-priority application bypasses the cache via disabled DDIO. Fig. 14 shows the 99^{th} percentile latency of the high-priority application–other percentiles show a similar trend with a smaller difference. These results demonstrate that bypassing cache via a remote socket (i.e., case iv) achieves the same latency as when there is no low-priority application (i.e., case i). However, when both applications are receiving traffic via DDIO (i.e., case ii), the 99^{th} percentile latency degrades ~30%. We observe that bypassing cache has the same benefits as increasing DDIO capacity (i.e., case iii vs. case iv). Furthermore, comparing cases (iv) and (v) indicates that disabling DDIO *slightly* pollutes the cache (as opposed to bypassing via a remote socket). We speculate that disabling DDIO only affects the packets, not the descriptors. Therefore, we conclude that bypassing cache can result in less variability in performance and potentially better performance isolation. Additionally, it is clearly necessary to tune DDIO capacity when moving toward 200 Gbps.



(a) Through local socket.

(b) Through remote socket.

Figure 13: Receiver setup to achieve 200 Gbps. On the right setup, the second NIC is connected to the remote socket. It sends packets through UPI link directly to the main memory.

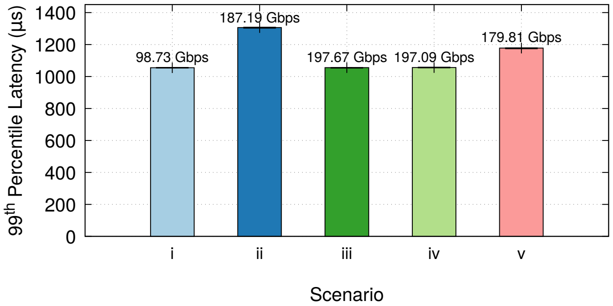


Figure 14: Bypassing cache and tuning DDIO at 200 Gbps mitigate I/O contention and improve the tail latency of the high-priority application up to 30%. Scenarios: (i) 100 Gbps with no contention; (ii) contention at 200 Gbps; (iii) tuning DDIO at 200 Gbps; (iv) bypassing cache via a remote socket; and (v) bypassing cache via disabled DDIO. The total achieved throughput of the receiver is written on the bars.

9 Lessons Learned: Optimization Guidelines

This section summarizes our key findings, which could help system designers/developers to optimize DDIO for their applications. Furthermore, our study should inspire computer architects to improve DCA's performance by offering increasing control. Although we focused on packet processing, our work is not limited to network functions. Our investigations could be equally useful in other contexts (e.g., HPC) that require high-bandwidth I/O when transferring data via RDMA and processing with GPUs. We showed that current approaches to avoid DDIO becoming a bottleneck are only temporary solutions and they are inapplicable to multi-hundred-gigabit network applications. We proposed a benchmarking method to understand the unknown & little-discussed details of DDIO. Later, we characterized the performance of DDIO in different scenarios and showed the benefits of bypassing the cache. We concluded that there is *no* one-size-fits-all approach to utilize DDIO. Our study reveals:

- The locations of LLC to which DDIO injects data (§4.1).
- Co-locating an application's code/data with I/O in the cache could adversely impact its performance (§4.2).
- The way that DDIO behavior changes for different system parameters and workload characteristics (§5).
- If an application is I/O bound, adding excessive cores could degrade its performance (Fig. 10).
- If an application is I/O bound, *carefully* sizing the DDIO capacity can improve its performance and could lead to the

same improvements as adding more cores (Fig. 11).

- If an application starts to become CPU bound, adding more cores can increase its throughput, but then it has to balance load among cores to *maximize* DDIO benefits (Fig. 12).
- If an application is *truly* CPU/memory bound, DDIO tuning is less efficient (Fig. 9). However, it can be beneficial to buffer in DRAM incoming requests/packets which cannot be processed in time, rather than having the NIC issue PAUSE frames or drop packets.
- Going beyond ~75 Gbps can cause DDIO to become a bottleneck (Fig. 8). Therefore, it is essential to *bypass cache* to realize performance isolation. Bypassing cache could be done for low-priority traffic or applications that do not benefit from DDIO (§8.1).
- Different applications have different levels of sensitivity to DDIO (§7). Identifying this level is essential to utilize system resources more efficiently, provide performance isolation, and improve performance.

10 Related Work

The most relevant work to our study is ResQ [83], which we discussed thoroughly in §3.1 and §8.1. This section discusses other efforts relevant to our work.

Injecting I/O into the cache. The idea of loading I/O data directly to the processor's cache was initially proposed using cache injection techniques [52, 63]. Later, it was used to enhance network performance on commodity servers and was referred to as DCA [25]. Amit Kumar et al. [42] investigated the role of coherency protocol in DCA. Their results indicated that the benefit of DCA would be limited when the network processing rate cannot match the I/O rate. In addition, [75] showed that DCA could cause cache pollution; hence they proposed an alternative cache injection mechanism to mitigate the problem. A. Kumar et al. [43] characterized DCA for 10-Gbps Ethernet links. Other works have discussed that DCA is insufficient due to architectural limitations [40, 46, 71]. For example, the work in [46] proposed a new I/O architecture that decouples and offloads I/O descriptor management from the NIC to an on-chip network engine. Similarly, the work in [40] proposed a flexible network DMA interface which can support DCA. Last but not least, Wen Su et al. [71] proposed an improvement to combine DCA with an integrated NIC to reduce latency.

Efforts toward realizing 100 Gbps. Many have tried to tackle challenges to achieve suitable performance for fast networks, mostly in the context of NFV [49] and key-value stores [19, 45]. Some research has exploited new features in modern/smart/programmable NICs (e.g., [38, 47, 84, 94]) & switches (e.g., [36]) or proposed new features (e.g., [70]) to offload costly software processing. A number of works investigate packet processing models (e.g., [9, 39, 93]). CacheBuilder [80] and CacheDirector [15] have discussed the importance of cache management in realizing 100-Gbps networks. HALO [92] exploited the non-uniform cache architecture (NUCA) characteristics of LLC to perform in-cache flow classification. Last but not least, IOctopus [68] proposed a new NIC design and wiring for servers to avoid non-uniform DMA penalties. Our work is complementary to these works.

Cache partitioning. Many have tried to overcome cache contention by performing cache partitioning [53]. These efforts can be split into two main categories: (i) software techniques and (ii) hardware techniques. The former group principally relies on physical addresses to partition cache based on sets [22, 48, 67] or slices [15]. This way of cache partitioning does not require any hardware support, but it is not very commonly used, due to its drawbacks (e.g., OS/App modification and costly re-partitioning). The latter group mostly exploits way-partitioning (e.g., CAT) to partition the cache among different applications [13, 18, 62, 89, 90, 91]. In addition to these techniques, Wang et al. [85] proposed a hybrid approach that combines both techniques to achieve finer granularity for partitioning. To the best of our knowledge, there are only two works (ResQ [83] and CacheDirector [15]) that have specifically tried to exploit cache partitioning techniques to improve packet processing. ResQ proposes to isolate a percentage of LLC that is used for I/O and CacheDirector exploits the NUCA used in Intel processors to distribute I/O more efficiently among different LLC slices. Our work is complementary to these works, as most of them do not consider I/O when partitioning the cache.

11 Conclusion

DCA technologies were introduced to improve the performance of networking applications. However, we systematically showed that the latest implementation of DCA in Intel processors (i.e., DDIO) cannot perform as needed with increasing link speeds. We demonstrated that better I/O management is required to meet the critical latency requirements of future networks. Our main goal is to emphasize that networking is, now more than before, tightly coupled with the capability of the current hardware. Consequently, realizing time-critical multi-hundred-gigabit networks is only possible by (i) increasingly well-documented control over the hardware and (ii) improved holistic system design optimizations.

Acknowledgments

We would like to thank our shepherd, Mark Silberstein, and anonymous reviewers for their insightful comments. We are grateful to Tom Barbette for helping us with his NPF tool. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work was also funded by the Swedish Foundation for Strategic research (SSF). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

References

[1] Direct Cache Access (DCA), Oct 2010. ftp://supermicro.com/ISO_Extracted/CDR-X8-Q_1.02_for_Intel_X8_Q_platform/Intel/LAN/v16.3/PROXGB/DOCS/SERVER/DCA.htm, accessed 2019-08-05.

[2] Memcached Benchmark, 2015. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark, accessed 2019-12-30.

[3] Benchmarking - Benchmarking Linux with Sysbench, FIO, Ioping, and UnixBench: Lots of Examples. https://wiki.mikejung.biz/Benchmarking, 2018.

[4] Flexible I/O Tester (fio). https://fio.readthedocs.io/en/latest/fio_doc.html, 2019.

[5] Seastar. http://seastar.io/, 2019.

[6] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. Data Direct I/O Characterization for Future I/O System Exploration. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020. https://yifanyuan3.github.io/publication/ddio_gem5, accessed 2020-05-20.

[7] Philip C Arellano and James A Coleman. Method, apparatus, and system for allocating cache using traffic class, March 30 2017. US Patent App. 14/866,862.

[8] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, Santa Clara, CA, February 2020. USENIX Association.

[11] Harsha Basavaraj. A case for effective utilization of Direct Cache Access for big data workloads. Master's thesis, UC San Diego, 2017. https://escholarship.org/uc/item/0fr3735b, accessed 2019-07-24.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[13] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, Feb 2018.

[14] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[15] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, 2019. ACM.

[16] Financial Services Industry (FSI) - Frequently Asked Questions. https://software.intel.com/en-us/articles/financial-services-industry-fsi-frequently-asked-questions, accessed 2019-07-24.

[17] Intel Forum. Intel Ethernet X520 to XL710 - Tuning the buffers: a practical guide to reduce or avoid packet loss in DPDK applications. https://etherealmind.com/wp-content/uploads/2017/01/X520_to_XL710_Tuning_The_Buffers.pdf, accessed 2019-07-24.

[18] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 295–308, Denver, CO, June 2016. USENIX Association.

[19] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[20] Jeff Gilbert and Mark Rowland. The Intel Xeon Processor E5 Family: Architecture, Power Efficiency, and Performance, August 2012. https://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-8-DataCenter/HC24.29.827-Xeon-Rowland-Xeon-E5-2600-Disclaimer.pdf, accessed 2019-07-24.

[21] Andrew Herdrich, Khawar Abbasi, and Marcel Cornu. Introduction to Memory Bandwidth Allocation, March 2019. https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation, accessed 2019-07-24.

[22] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 23–32, July 2011.

[23] How to disable Data Direct I/O (DDIO) on Intel Xeon E5? https://forums.intel.com/s/question/0D50P0000490NFhSAM/how-to-disable-data-direct-io-ddio-on-intel-xeon-e5?language=en_US, accessed 2019-07-24.

[24] Ted Hudek. Introduction to Receive Side Scaling, 04 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, accessed 2019-12-29.

[25] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, June 2005.

[26] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The Case for a Network Fast Path to the CPU. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 52–59, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Information about PCM PCIe counters. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/543883, accessed 2019-07-24.

[28] Intel. Intel Data Direct I/O Technology Overview, 2012. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html, accessed 2019-07-26.

[29] Intel. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring, July 2017. https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-uncore-performance-monitoring-manual.html, accessed 2019-07-26.

[30] Intel. Intel Arria 10 Avalon-ST Interface with SR-IOV PCIe Solutions User Guide, 2019. https://www.intel.com/content/www/us/en/programmable/documentation/lbl1415123763821.html#lbl1453336559194, accessed 2019-07-26.

[31] Intel Ethernet Flow Director and Memcached Performance, 2014. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf, accessed 2019-09-09.

[32] IO Issues: Remote Socket Accesses. https://software.intel.com/en-us/vtune-amplifier-cookbook-io-issues-remote-socket-accesses, accessed 2019-09-01.

[33] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge, 2012. https://bit.ly/2LKVfZr, accessed 2019-07-24.

[34] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[35] Muthurajan Jayakumar. Data Plane Development Kit: Performance Optimization Guidelines. https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper, accessed 2019-07-24.

[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[38] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation (NSDI 18)*, NSDI'18, pages 171–186, Renton, WA, 2018. USENIX Association.

[39] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr., and Dejan Kostić. SNF: synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2:e98, November 2016.

[40] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. *SIGPLAN Not.*, 51(4):67–81, March 2016.

[41] Maciek Konstantynowicz, Patrick Lu, and Shrikant M. Shah. Benchmarking and Analysis of Software Data Planes. Technical report, Cisco, Intel Corporation, FD.io, Dec 2017. https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf, accessed 2019-07-24.

[42] A. Kumar and R. Huggahalli. Impact of Cache Coherence Protocols on the Processing of Network Traffic. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 161–171, Dec 2007.

[43] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of Direct Cache Access on multi-core systems and 10GbE. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 341–352, Feb 2009.

[44] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *S&P*, May 2020. Intel Bounty Reward.

[45] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst.*, 34(2):5:1–5:30, April 2016.

[46] G. Liao, X. Znu, and L. Bnuyan. A new server I/O architecture for high speed networks. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 255–265, Feb 2011.

[47] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[48] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008.

[49] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of Performance Acceleration Techniques for Network Function Virtualization. *Proceedings of the IEEE*, 107(4):746–764, April 2019.

[50] Patrick Lu. Performance Considerations for Packet Processing on Intel Architecture, May 2017. https://fdio-vpp.readthedocs.io/en/latest/events/Summits/FDioMiniSummit/OSS_2017/2017_05_10_performanceconsideration.html, accessed 2019-07-24.

[51] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[52] V. Milutinovic, A. Milenkovic, and G. Sheaffer. The cache injection/cofetch architecture: initial performance evaluation. In *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 63–64, Jan 1997.

[53] Sparsh Mittal. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.*, 50(2):27:1–27:39, May 2017.

[54] Jeffrey C. Mogul and John Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *Proc. 17th Workshop on Hot Topics in Operating Systems (HoTOS)*, 2019.

[55] David Mulnix. Intel Xeon Processor Scalable Family Technical Overview, Sep 2017. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, accessed 2019-07-24.

[56] NetApp. What is the potential impact of PAUSE frames on a network connection?, Nov 2017. https://ntap.com/2RpAx1Q, accessed 2019-07-24.

[57] Network Performance Framework. https://github.com/tbarbette/npf, accessed 2019-07-24.

[58] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018*

Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. ACM.

[59] Khang Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, Feb 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, accessed 2019-07-24.

[60] Khang T Nguyen. Code and Data Prioritization - Introduction and Usage Models in the Intel® Xeon® Processor E5 v4 Family, 2016. https://software.intel.com/en-us/articles/introduction-to-code-and-data-prioritization-with-usage-models, accessed 2019-07-26.

[61] John Ousterhout. Always Measure One Level Deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[62] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 10:1–10:16, New York, NY, USA, 2019. ACM.

[63] Hazim Shafi Patrick Joseph Bohrer, Ramakrishnan Rajamony. Method and apparatus for accelerating input/output processing using cache injections , March 2004. US Patent No. US6711650B1.

[64] PCIe Bandwidth Drops on Skylake-SP. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/741386, accessed 2019-07-24.

[65] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[66] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, April 2016.

[67] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 155–164, New York, NY, USA, 1999. ACM.

[68] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. IOctopus: Outsmarting Nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 101–115, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Splash-3 Benchmark Suite. https://github.com/SakalisC/Splash-3, accessed 2019-07-24.

[70] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, February 2019. USENIX Association.

[71] W. Su, L. Zhang, D. Tang, and X. Gao. Using Direct Cache Access Combined with Integrated NIC Architecture to Accelerate Network Processing. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 509–515, June 2012.

[72] Roman Sudarikov and Patrick Lu. Hardware-Level Performance Analysis of Platform I/O, June 2018. https://dpdkprcsummit2018.sched.com/event/EsPa/hardware-level-performance-analysis-of-platform-io, accessed 2019-07-24.

[73] Supermicro. *1028UX-LL1-B8, 1028UX-LL2-B8, and 1028-LL3-B8 User's Manual*. https://www.supermicro.com/manuals/superserver/1U/MNL-1886.pdf, accessed 2019-07-24.

[74] Supermicro. *6028UX-TR4 User's Manual*. https://www.supermicro.com/manuals/superserver/2U/MNL-1706.pdf, accessed 2019-07-24.

[75] D. Tang, Y. Bao, W. Hu, and M. Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[76] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel, 2019. https://arxiv.org/pdf/1909.04841.pdf, accessed 2019-09-15.

[77] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Observing Network Packets over a Cache Side-Channel. In *Proceedings of the 47th*

International Symposium on Computer Architecture, ISCA '20, New York, NY, USA, 2020.

[78] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *CoRR*, abs/1810.09360, 2018.

[79] Temporary PCIe Bandwidth Drops on Haswell-v3. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/600913, accessed 2019-07-24.

[80] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. Dark Packets and the End of Network Scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, pages 1–14, New York, NY, USA, 2018. ACM.

[81] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[82] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 21:1–21:16, New York, NY, USA, 2019. ACM.

[83] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, Renton, WA, April 2018. USENIX Association.

[84] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[85] X. Wang, S. Chen, J. Setter, and J. F. Martínez. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, Feb 2017.

[86] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization, Jan 2017. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, accessed 2019-07-24.

[87] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, accessed 2019-07-24.

[88] Xeon E5 disable DDIO in OS? https://forums.intel.com/s/question/0D50P0000490VP0SAM/xeon-e5-disable-ddio-in-os?language=en_US, accessed 2019-07-24.

[89] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 13:1–13:15, New York, NY, USA, 2018. ACM.

[90] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 14:1–14:13, New York, NY, USA, 2018. ACM.

[91] M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic Cache Management Using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

[92] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 601–614, New York, NY, USA, 2019. ACM.

[93] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A Closer Look at NFV Execution Models. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, pages 85–91, New York, NY, USA, 2019. ACM.

[94] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sep. 2014.

sRDMA — Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access

Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler
Department of Computer Science, ETH Zurich

Abstract

State-of-the-art remote direct memory access (RDMA) technologies have shown to be vulnerable against attacks by in-network adversaries, as they provide only a weak form of protection by including access tokens in each message. A network eavesdropper can easily obtain sensitive information and modify bypassing packets, affecting not only secrecy but also integrity. Tampering with packets can have drastic consequences. For example, when memory pages with code are changed remotely, altering packet contents enables remote code injection. We propose sRDMA, a protocol that provides efficient authentication and encryption for RDMA to prevent information leakage and message tampering. sRDMA uses symmetric cryptography and employs network interface cards to perform cryptographic operations. Additionally, we provide an implementation for sRDMA using programmable network adapters.

1 Introduction

Despite numerous state-of-the-art systems [8, 11, 30] leveraging remote direct memory access (RDMA) primitives to achieve high performance guarantees and resource utilization, current RDMA technologies lack any form of cryptographic authentication or encryption. Instead RDMA mechanisms provide a weak form of protection by including access tokens in each message. Given that RDMA networks are mainly used in data-center environments and at large-scale deployments, detecting bugged wires is seemingly impossible. But not only in-network adversaries are an issue, also malicious end hosts can affect the security of an RDMA network. If an adversary is able to obtain control over a machine in an RDMA network (e.g., by escaping its virtual machine or hypervisor confinement in a cloud service [42]), it can fabricate and inject arbitrary packets. If the adversary can guess or obtain the protection domain and memory protection tokens (which are transmitted in plaintext), it can read and write memory locations that have been exposed using RDMA on *any machine in the network*, leading to a powerful attack vector for lateral movement in a data center network.

Given these threats, the security of current RDMA data center networks highly depends on isolation. However, even isolation cannot defend against in-network attackers. Thus, RDMA networks require cryptographic authentication and encryption. Unfortunately, application-level encryption (e.g., *TLS* [34]) is not possible, since RDMA read and write can operate as purely one-sided communication routines. Furthermore, such an approach requires employing a temporal buffer for incoming encrypted messages. The message would then be decrypted by the CPU and copied to the desired location, which would cause high overhead—negating RDMA's advantages. Additionally, cryptographic protection using *IPSec* [10] does not support RDMA traffic as the protocol is unaware of the underlying RDMA headers and achieves no source authentication (see Section 7).

In our work, we introduce a secure RDMA (sRDMA) design using a secure reliable connection (SRC) queue pair (QP) that uses symmetric cryptography for source and data authentication and employs Network Interface Cards (NICs) to perform cryptographic operations. Symmetric cryptography reduces the computational overhead compared to asymmetric cryptography by 3–5 orders of magnitude. Thus, it is suitable for high-performance and low-latency applications based on RDMA, e.g., [8, 17, 39]. Since symmetric cryptography introduces per-connection memory overhead and memory on NICs is constrained, we augment our proposed mechanisms using protection domain level keys and efficient dynamic key derivation, which eliminates the need for storing QP-level keys and drastically reduces the memory overhead on RDMA-capable NICs (RNICs). Additionally, we propose extended memory protection mechanisms that enable delegation of memory access to other trusted peers without requiring additional communication with the accessed host (e.g., an access control proxy for databases [29]).

In summary, we make the following contributions:

- we design a SRC QP that effectively prevents attacks in an RDMA network, with minimal changes to the current

InfiniBand Architecture (IBA) standard (Section 4.2);

- we improve our design by introducing PD-level keys to reduce the memory overhead on the RNIC (Section 4.5), and augment IBA with extended memory protection that enables delegation of memory accesses to third parties without the need of direct communication to the target entity (Section 4.6);
- we provide an implementation of our design using modern programmable network adapters equipped with ARM multi-core processors [7, 40] (Section 5);
- we extensively evaluate our design using artificial and real-world traces. Additionally, we modified the RDMA-based key value store, HERD [17], to make use of sRDMA (Section 6).

2 Remote Direct Memory Access

RDMA is a mechanism allowing one machine to directly access data in remote machines across the network.

Memory accesses are performed using dedicated hardware without any CPU intervention or context switches, which decreases CPU usage on both the initiator and the target. When an application performs an RDMA read or write request, the application data is delivered directly to the network, reducing latency and enabling fast message transfer. RDMA also exhibits the concept of one-sided operations when the CPU at a target node is not notified of incoming RDMA requests.

Several network architectures support RDMA: InfiniBand (IB) [4] , RDMA over Converged Ethernet (RoCE) [5], and internet Wide Area RDMA Protocol (iWARP) [33]. InfiniBand is a network architecture fully designed to enable reliable RDMA with its own hardware and protocol specification. RoCE is an extension to Ethernet to enable RDMA over an Ethernet network. Finally, iWARP is a protocol that allows using RDMA over TCP/IP. In this work, we focus on the InfiniBand architecture (IBA) and RoCE as they are the most widely used interconnect for RDMA, but the proposed ideas can be easily extended to other RDMA architectures.

2.1 InfiniBand Transport

Several transport types are supported by the IBA to communicate between endpoints: reliable connection (RC), unreliable connection, unreliable datagram, extended reliable connection, and raw packet. In this paper, we only consider the RC transport type, since it is the only type that supports both RDMA read and write requests.

The RC transport type establishes a *queue pair (QP)* between the two communicating parties. QPs are bi-directional message transport engines used to send and receive data in InfiniBand. Endpoints of a single RC QP can only communicate with each other but not with any other QP in the same or any other target adapter. Each QP endpoint has a queue pair number (QPN) assigned by the RNIC which uniquely identifies the QP within the RNIC.

The RC transport uses several techniques to ensure reliability. The target must respond to each request packet with a positive acknowledge packet or a negative acknowledge packet. The acknowledgement-based protocol permits the requester to verify that all packets are delivered to the target. To ensure the integrity of a packet, each packet contains two checksums that are verified by the target node. Finally, the RNIC counts received and sent packets using a packet sequence number (PSN), which is included in each packet. Thus, endpoints of a QP must know the PSN of each other to enforce in-order delivery and detect duplicate and lost packets.

2.2 IBA Memory Protection

The IBA protection mechanisms provide protection from unauthorized access to the local memory by network controllers. The local memory can also be protected against prohibited memory accesses. Three mechanisms exist to enforce memory access restrictions: Memory Regions, Protection Domains (PD), and Memory Windows.

Memory Regions. To get access to host memory, the RNIC must first allocate a corresponding memory region. This process involves copying page table entries of the corresponding memory to the memory management unit of the RNIC. When a memory region is created, the RNIC generates keys for local and remote accesses, namely *l_key* and *r_key*. The memory region can be accessed by any local QP which has the *l_key* as long as they are in the same PD, and by their remote QP endpoints which have the *r_key*. The endpoints must prove the possession of this key by including it in every RDMA request, such as RDMA Write and Read. *r_key* is not used in any form of cryptographic computation, but rather is used as access tokens that are transmitted in plaintext.

Protection Domain. PDs provide protection from unauthorized or inadvertent use of memory regions. PDs group IB resources such as QP connections and memory regions that can work together: QP connections created in one PD cannot access memory regions allocated in another PD. In other words, a memory region can be accessed by any QP from its PD. All QPs and memory regions must have a PD and can be a member of one PD only.

Memory Windows. Memory windows extend protection of memory regions by allowing remote QPs to have different access rights within a memory region and grant access to only a slice of the memory region.

3 Problem Definition

This section describes the adversary model we consider, outlines different types of attacks, and the security properties we strive to achieve.

3.1 Desired Security Properties

The current IBA protection mechanisms do not suffice to ensure secure communication between endpoints, allowing adversaries numerous attacks. Thus, the primary goal of our work is to *secure RDMA protocols against attacks* by providing source and data authentication along with data secrecy and data freshness. Source authentication denotes the verification of the source address of a host that sends a packet and is designed to determine whether a packet originated from the claimed source. Data authentication ensures that the packet content has not been modified. Data secrecy ensures that the data remains hidden from a network eavesdropper. Data freshness ensures that data has not been recorded and replayed by a network attacker. Additionally, our proposal should require *minimal changes to the protocol*, and introduce only a *minor performance overhead*. This does not only include latency and processing overhead for RDMA requests but also memory state overhead on the RNIC.

3.2 Adversary Model

In our adversary model we consider end hosts that are equipped with RNICs and interact with each other through RDMA, and an adversary with the following parameters.

Location. We assume that the adversary can reside at *arbitrary locations* within the network. Thus, we consider both network-based adversaries (e.g., rogue cloud provider, rogue administrator, malicious bump-in-the-wire device) and adversaries located at end hosts (e.g., compromise of an end host). This includes compromise of the machines of communicating parties. However, we assume that RNICs are *trusted* by their host. This could be achieved using remote attestation, whereby a trusted party checks the internal state of a potentially compromised network device. We further assume that the internal bus is trusted, such that the CPU can securely communicate with the RNIC.

Capabilities. A network-based adversary can passively eavesdrop on messages, but also actively tamper with the communication. Since RDMA communication is performed in plaintext, an adversary that is located on the path between communicating parties can obtain any information in all IB and Ethernet headers. Furthermore, he can also alter any of these values, as this only requires recalculation of packet checksums, whose algorithms and seeds are known and specified by the IBA.

Given these capabilities, the adversary can also fabricate packets and send them towards a destination of its choice using spoofed QP numbers, *r_key*s, and PSNs (e.g., to modify a memory region without authorization to influence the behavior of applications running on the remote host).

Cryptography. The adversary has no efficient way of breaking cryptographic primitives. For pseudorandom function families, this means that no efficient algorithm can distinguish between an output of a function chosen randomly from the pseudorandom function family (PRF) and a random value.

Table 1: Notation used in this paper.

$\|\|$	Bitstring concatenation
$PRF_K(\cdot)$	Pseudorandom function using key K
$MAC_K(\cdot)$	Message authentication code using key K
A, B	Endpoints uniquely identified by the combination of the adapter port address (APA) and Queue Pair Numbers (QPN)
$K_{A,B}$	Symmetric key shared between node A and B
$nonce_{A \rightarrow B}$	cryptographic nonce used for communication in the direction from node A to node B
K_{PD}, K_{MR}, K_{SR}	Symmetric key used for protection domain, memory region, or sub memory region

4 Secure RDMA System Design

We propose a new transport type for reliable communication based on the IBA. We introduce a secure reliable connection (SRC) QP that uses symmetric cryptography for source and data authentication, and thus provides guarantees for the origin of a packet, data authenticity and payload secrecy.

To require minimal changes to the current IBA specification, our proposed design of the SRC QP consists of two main changes: 1) we add symmetric key initialization for QPs, and 2) we propose a new packet header called secure transport header (STH) which contains a message authentication code (MAC) providing integrity of the packet content. The STH must be included in all requests and response packets corresponding to RDMA reads and writes.

Besides basic QP-channel protection, we also propose *PD level protection* eliminating the need for storing cryptographic keys for each QP, which drastically reduces the memory overhead on RNICs. Additionally, it enables *extended memory protection* that provides memory access control based on encryption and the ability of delegating memory access to other trusted entities without additional communication to the accessed host. All QPs and memory regions created in a secure PD will be inherently secured by it.

Table 1 lists the security-related notation used in this paper.

4.1 Assumptions

Trust in RNIC. We assume that the RNIC is trusted by its host. It can not only perform authentication of outgoing packets, but is also trusted to perform en-/decryption of the packet payload. We further assume that the internal bus is trusted, such that the CPU can securely communicate with the RNIC.

QP-level Key Establishment. Our system enables the establishment of a QP-level symmetric key. To guarantee interoperability, our design is agnostic of this underlying mechanism. IBA could use for instance a (D)TLS [34] or QUIC [15] handshake as a mechanism to obtain a QP-level symmetric key.

Key Validity. As the validity period of a QP-level symmetric key is bound to the lifetime of a QP, key rollover can be performed by closing and reopening a QP between the communicating entities. Thus, key lifetime can be managed on the application level.

Size (bits)	0	96	128	160	224	256	384	512
Value	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7

Table 2: Possible sizes of STH, depending on the 3 bit value indicated in the Base Transport Header of an IB packet.

4.2 Secure Reliable Connection Queue Pair

We propose new transport type—Secure Reliable Connection (SRC) QP—that uses symmetric cryptography for source and data authentication. The introduction of SRC requires minimal changes to the current specification. Specifically, the QP initialization requires specifying a protection algorithm and a symmetric key. This allows us to bootstrap secrecy and authentication for QP-based communication.

Secure Transport Header. Secure Transport Header (STH) consists of MAC to provide header and packet authentication. The STH must be included in all request and response packets of sRDMA. Depending on the authentication mode installed to the secure QP, the MAC either authenticates only the packet header or the entire packet. To specify the length of the STH, we use 3 (out of 7) reserved invariant bits in the Base Transport Header. Based on the 3 bit value (see Table 2), the size of the MAC changes: minimum 96 bits, and maximum 512 bits. If the reserved 3 bits are all zero, then the STH is not present in the packet, thereby enabling support of both classical and secure QP connections.

Reusing PSN as a Per-Packet Nonce. sRDMA prevents replay attacks by including a unique nonce in the MAC computation of each packet (Section 4.3). Nonces are used as initialization vectors for ciphers to ensure that every packet is unique. They must *only be used once*, but their choice can be *predictable* and they can be *transmitted in clear* [19, 37]. In case a nonce is reused, the cryptographic properties of a cipher are affected (e.g., "sudden death" property of Poly1305 [6]).

A naive solution is to transmit a nonce as cleartext with each packet (e.g., as in TLS up to version 1.2 [36]). However, this would incur an additional transmission overhead of at least 64 bits, and additional 64 bits memory overhead on RNICs memory to store the nonce.

To avoid the overhead of transmitting the nonce, our protocol takes advantage of the sequential nature of IB packets. It uses the sequence number as nonces as they are tracked by end points and can never be reused. The approach resembles how TLS 1.3 [35] exploits the packet number as a nonce; however, the size of the PSN in the IB packet is only 24 bits, which would cause a reuse of a nonce after 80 ms assuming that an RNIC is able to send 200 million packets per second [28].

sRDMA extends the local PSN counters for inbound and outbound packets on the RNIC to 64 bits each, and reuse them as a per-packet nonce, thereby introducing only 40 bits overhead for each nonce. However, the size of the PSN transmitted on the wire remains unchanged (24 bits) and contains the least significant bits of the 64 bit counter. sRDMA is able to infer the 64 bit nonce used to secure the packet using only the 24 bit PSN specified in the header. Under the same assumption on the packet rate, the reuse of nonce occurs after 3,000 years.

Table 3: Overheads of sRDMA for *N* RC QP connections with AES-128 cipher in 4 different protection modes. Here, *pd-prot* and *ext-mem* stand for PD-level protection and extended memory protection, and are described in Section 4.5 and 4.6.

AES-128 protection	Key overhead	Nonce	Header
basic	16B * *N*	10B * *N*	16B
pd-prot	16B	10B * *N*	16B
ext-mem	16B * *N* + 16B	10B * *N*	16B
pd-prot + *ext-mem*	16B + 16B	10B * *N*	16B

To ensure that the nonce never gets reused by both endpoints, we use the most significant bit to identify the direction of communication between the entities *A* and *B* using their endpoint identifiers: the combination of adapter port address and Queue Pair Number (QPN).

4.3 Header Authentication

To perform header authentication, sRDMA uses the established symmetric keys and calculates a MAC for each packet:

$$mac_{hdr} = MAC_{K_{A,B}}(nonce_{A\to B} \parallel RH \parallel BTH)$$

Here, *RH* denotes the routing header, which defines the adapter port address, and *BTH* the base transport header, which includes destination QPN. Note that these headers uniquely identify the sender and receiver RNIC, and limit the input size of the MAC computation (only the packet header instead of the entire packet with arbitrary payload length). Thus, assuming an block-cipher-based MAC is used, a fixed number invocations of the block-cipher are required to calculate a MAC.

The RNIC of the receiving node will recompute the MAC for each packet and compare it to the MAC appended in the STH. Fields that are modified during the packet's transmission are replaced with ones during the MAC computation (same as for invariant checksum).

Header authentication prevents not only source-address-spoofing attacks, but also unauthorized access to memory regions by augmenting the existing IBA memory-protection mechanisms (i.e., *r_key* and memory windows).

4.4 Packet Authentication and Encryption

For packet authentication and payload encryption, we assume that the RNIC is trusted. Thus, the host is allowed to offload all cryptographic operations to the RNIC. We use authenticated encryption with associated data (AEAD), to simultaneously obtain secrecy and authenticity for the payload. The authentication tag is transmitted using the MAC field in the STH.

4.5 PD-level Protection

Introducing QP-level keys requires storing a 16 byte key per QP (see Table 3). As an RNIC might have a large number of QPs simultaneously, this can lead to a significant memory overhead on the RNIC. Memory on RNICs is a constrained resource, and a large part is consumed by IB connection contexts and page-table entries for registered memory. Multiple works report significant performance degradation of RDMA operations when the amount of memory registered or the number of QPs is increased [11, 18]. This is due to the RNIC running out of memory for storing page-table entries and starting to fetch them from system memory across the PCI bus. For instance, Dragojevic et al. [11] observe ~4x throughput drop in their evaluation when 4,096 memory pages are registered within the RNIC compared to a single-page experiment. Thus, we aim to mitigate the memory overheads introduced by QP-level keys.

To reduce the memory overhead and eliminate the need of storing a symmetric key per QP, we introduce PD-level protection. In this mechanism, we assign a symmetric key K_{PD} to each protection domain PD and use this key to derive QP-level keys using efficient key derivation [14]. PD-level keys are exchanged using the same mechanism as QP-level keys (see Section 4.1). The derivation process works as follows:

$$K_{A,B} = PRF_{K_{PD}}(APA_A \parallel QPN_A \parallel APA_A \parallel QPN_B)$$

PRF denotes a pseudorandom function with a PD-level key K_{PD} and a pair of unique end point identifiers (i.e., adapter port address (APA) and queue pair number (QPN)) as input. When an RDMA request targets a QP that is located within a protection domain PD, the RNIC uses the corresponding symmetric key K_{PD} to derive the QP-level key on-the-fly. The QP-level key is then used to perform authentication and encryption. Thus, instead of storing a symmetric key per QP, the RNIC is only required to store a key per PD. To minimize the processing overhead, the RNIC can cache the derived QP-level keys (e.g., after the first packet of a message arrives). K_{PD} is initialized upon creation of the PD and thus the lifetime of K_{PD} is bound to the lifetime of the PD. In order to perform a key rollover, a new protection domain must be created.

4.6 Extended Memory Protection

Using encryption of memory regions enables an even stronger mechanism for access control, as only entities in possession of the required key are able to read the content of a memory region. For this purpose, we use PD-level memory protection and derive memory level keys from K_{PD} for memory regions that are created within the protection domain. The derivation process works as follows:

$$K_{MR} = PRF_{K_{PD}}(START_{MR} \parallel END_{MR} \parallel r\_key_{MR})$$

Alternatively, the K_{MR} can be provided by the application to protect memory from unauthorized accesses.

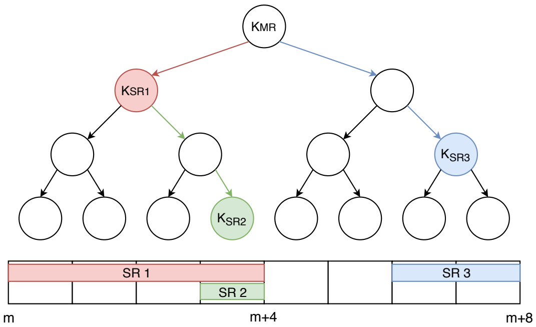


Figure 1: Access Sub-Delegation with one-way tree.

When remote parties want to access a subregion (SR) of the region MR, they need to prove the possession of the K_{MR} by computing a key to the SR:

$$K_{SR} = PRF_{K_{MR}}(START_{SR} \parallel END_{SR}) \quad (1)$$

Nonce for Key Derivation. To avoid replay attacks, our system must use a separate nonce for each memory region. However, it is not possible to use a memory access counter as nonce, as multiple QPs can access the same memory region. Therefore, this would require the RNIC to include a random nonce in each packet, which must be unique among all nonces used to access the memory region. Given that multiple parties have access to the region, this property is hard to achieve. Additionally, we want to avoid transferring a separate MAC for memory access in the packet header. Thus, we suggest to reuse the MAC of the header by overwriting it as follows:

$$mac_{hdr} = MAC_{K_{A,B}}(K_{SR}||mac_{hdr})$$

Such design allows sRDMA to reuse the per-packet nonce used in computation of mac_{hdr} and ensure the possession of K_{MR} to access memory. This construction is secure since the key is unknown to an adversary.

4.7 Sub-Delegation of Access to Memory

To allow sub-delegation of access to memory regions, we further extend the proposed extended memory protection with a binary one-way function tree [26]. The one-way function tree is built top-down where the memory region key K_{MR} is represented by the root of the binary tree and all child nodes are generated by applying the PRF:

$$K_{MR_{child}} = PRF_{K_{MR_{parent}}}(START_{MR_{child}} \parallel END_{MR_{child}})$$

Each memory block represents a leaf of the binary one-way tree (see Figure 1). Thus, the height of the tree depends on the size of the region and on the memory block size. Delegating access to a subregion works by sending the key of an intermediate node to a remote party. Given the key for a memory region, the subregion offset and subregion size uniquely identify which inner node is required for delegation.

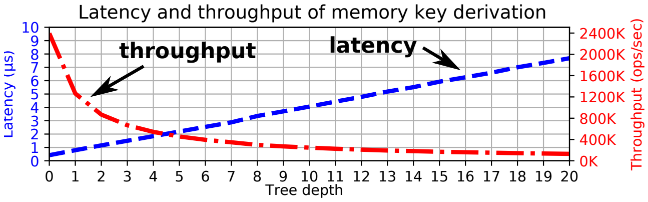


Figure 2: Performance of K_{MR} derivation for different tree depths. The derivation is performed sequentially, thus latency grows linearly and the throughput decreases exponentially.

An example of the delegation process is illustrated in Figure 1. To obtain a key for subregion 3, the entity must derive an intermediate key first, which can then be used to derive K_{SR_3}. To delegate access for subregion 2 to another host, a host in possession of K_{SR_1} derives K_{SR_2} and then shares this key with the delegated host. This allows access to subregion 2, but not to any other part of the memory region.

Only logarithmically many derivations are needed to obtain a key for memory access. For example, a key for accessing a 1 MiB subregion of a 16 MiB memory region can be obtained in at most 4 steps, which takes 2 μs on our RNIC (see Figure 2). Additionally, to prevent long key derivation chains, sRDMA allows users to limit the number of steps in the key derivation process. The sub-delegation is optional and can be disabled by restricting the tree depth to zero. In this case, the key derivation works as in Equation 1.

Advantages. This approach offers multiple advantages: 1) It trivially enables sub-delegation, as the delegated party can also calculate inner nodes of the subtree that is rooted at the subregion key. 2) The block size and thus the tree size is variable and can be adjusted for each memory region, e.g., depending on window size, depth of sub-delegation, or computational power of the corresponding nodes. This also allows limiting the depth of the tree to restrict the computational overhead. 3) The RNIC is required to compute only a single branch of the tree to verify. 4) The larger the delegate memory space, the smaller is the computational overhead on the RNIC. 5) The packet size remains constant as accessing a memory subregion requires only the start and size of the region, the offset and the size of subregion, and a MAC computed using the appropriate key.

Restrictions. We restrict delegation to powers of two to ensure that a single key is sufficient to delegate access to any sub-region. Alternatively, a solution based on segment trees would overcome this limitation, but require the exchange of multiple keys.

5 Implementation

Towards our goal of supporting secure QP connections, this section describes how we implement the sRDMA protocol using modern programmable network adapters equipped with ARM multi-core processors [7,40]. sRDMA core primitives are implemented in 3,500 lines of C++ code and rely on various libraries: libibverbs, an implementation of the RDMA verbs; librdmacm, an implementation of the RDMA connection manager; Openssl 1.1.1a, a general-purpose cryptography library; and libev, a high-performance event loop. Our implementation supports more than 20 different cryptographic algorithms, such as the AES cipher and SHA hash families, to enable authentication and data secrecy for secure QPs. The implementation, all tests, and benchmark scripts are available in the open-source release.*

5.1 Notation and Experimental Setup

In the rest of the paper, we refer to a programmable network adapter as a *SmartNIC*. Our SmartNIC is capable of running a full Linux stack, supports RDMA over RoCEv2 and has crypto acceleration enabled. When RDMA requests are initiated on the SmartNIC and target the local host we refer to them as DMA requests as they only pass across the PCIe bus.

Our implementation is bi-directional, i.e., sRDMA writes and reads can be sent in both directions passing through both SmartNICs of the initiator and the target. Therefore, we distinguish between three roles as depicted in Figure 4: *Initiator*, *SmartNIC*, and *Target*, and the initiator always communicates with the target via two SmartNICs. Such a design allows full offloading of cryptographic computations from the initiator and the target to their respective SmartNICs.

5.2 Implementation of the Secure QP

We provide a library that models a secure QP connection between an initiator and a target as three standard RC QPs: one DMA connection between the SmartNIC and the host on *each* endpoint, and one connection between the SmartNICs.

Connection Establishment. Our secure QP library encapsulates connection establishment, which is performed in three stages as for a classical RC QP. When an application wants to establish a secure QP, it first creates a local QP in the INIT state. In this state, the connection between the host and the SmartNIC is created, and all necessary symmetric keys are copied to the SmartNIC. Then the QP must be transitioned to the RTR state by passing information about the target such as the QPN, the LID, and the PSN. To perform this transition we establish an RC QP connection between the two SmartNICs and create a special connection context on each SmartNIC. Finally, to send messages we transit the secure QP to the RTS state by passing the local send PSN. The application workers on the SmartNIC are responsible for packet counting, key derivation, and cryptographic algorithms.

Memory Registration. When a memory region should be secured with extended memory protection, the library intercepts a memory registration request and sends memory region information to a thread on the local SmartNIC.

*https://spcl.inf.ethz.ch/Research/Security/sRDMA/

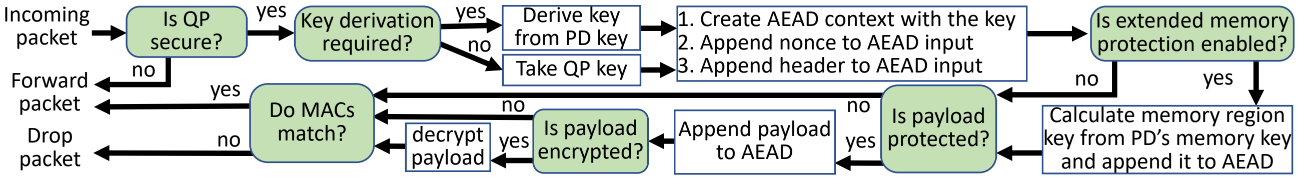
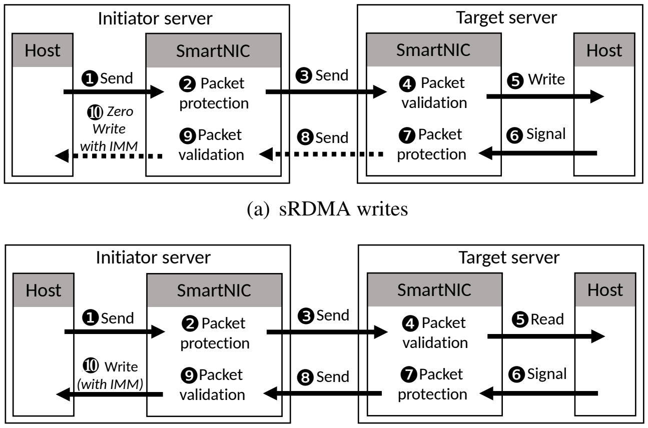


Figure 3: RNIC packet processing on receive.

Secure QP communication. The initiator uses IB Send to deliver packets for both sRDMA reads and writes to its SmartNIC. The SmartNIC uses IB Receive to receive incoming packets from DMA connections and from remote SmartNICs. The SmartNIC secures all incoming packets from a DMA connection according to the cryptographic mechanism agreed on with the target. To secure a packet, the SmartNIC appends the IB transport and RDMA headers along with the generated MAC to the packet header. In our implementation, we use IB scatter/gather entries to attach an additional header before the main payload provided by the initiator. Scatter/gather entries allow building up an outgoing message from multiple buffers. After that, the packets are forwarded to the SmartNIC of the target QP. The target's SmartNIC verifies the security header as depicted in Figure 3, and decides on initiating an RDMA Read or RDMA write depending on the type of the request towards the target's host. The replies from the target are secured by its SmartNIC and forwarded back to the initiator.

sRDMA request completion. If the initiator expects an acknowledgment for a signaled request, the SmartNIC is responsible for acknowledging the initiator about the completion of the request. We use IB requests with immediate data to generate completion events on the host. The secure QP library is able to intercept completion events to distinguish between classical IB completions and sRDMA completions. The intercepted sRDMA completions are modified to inform the initiator about the sRDMA completion instead of the classical IB completion.

Packet security. The whole process of packet verification and key generation is shown in Figure 3. The SmartNIC performs header authentication, packet authentication, or payload encryption depending on which security protocol has been set up for the QP and which packet is processed. The SmartNIC will derive the QP's key if the QP is initialized in a secure PD, and also verify extended memory protection if the registered memory region has extended memory protection set up. On receiving, the SmartNIC also checks whether the QP is indeed a secure QP, as our implementation also supports classical insecure RC QPs. For insecure RC QPs, packets do not carry a MAC and are always trusted by SmartNICs.



(a) sRDMA writes

(b) sRDMA reads

Figure 4: Implementation of RDMA operations.

5.3 sRDMA requests

Figure 4(a) depicts the implementation of an sRDMA write. The initiator ❶ sends a packet to the local SmartNIC containing the payload and the remote memory address. The local SmartNIC ❷ appends the IB header and the STH and ❸ sends the secured packet to the remote SmartNIC. The remote SmartNIC ❹ processes the header and ❺ initiates a signaled DMA write to the host memory specified in the header. Upon ❻ the completion of the DMA write, the SmartNIC, depending on whether the sRDMA write is signaled, ❼❽ sends an authenticated Ack packet to the initiator's SmartNIC. The SmartNIC of the initiator then ❾ verifies the packet and ❿ performs an empty RDMA write with immediate data to its host, which consumes one posted receive at the host application. Finally, the secure QP interface intercepts such completions and modifies them to notify the application about the secure request completion.

sRDMA also implements secure Send operations which are similar to sRDMA writes, but they always generate the completion on the target and do not require knowing destination buffers. Since a Send request does not contain the header with destination buffer, it does not support memory protection.

sRDMA read has a similar structure as an sRDMA write as depicted in Figure 4(b) but there are some subtle differences. The initiator ❶ sends the message containing remote and local memory addresses and their *r_key*s to the local SmartNIC. The initiator's SmartNIC creates a special local read completion context with the initiator's memory address where the remote data must be copied to. Then the local SmartNIC ❷❸ sends the authenticated read request to the remote SmartNIC, which ❹ verifies the request and ❺ initiates a signaled DMA read from the target host memory to one of the SmartNIC's buffers. When ❻ the completion of a DMA read is generated, the SmartNIC ❼❽ sends an authenticated read response with read data to the initiator's SmartNIC. The initiator's SmartNIC ❾ verifies the MAC of response packets and decides whether to ❿ write their content to the memory address specified in the matched local read completion context using a DMA write request. The DMA write will be with immediate data if the sRDMA read is signaled.

6 Evaluation

We conduct a series of benchmarks to thoroughly profile our system. To evaluate the overall sRDMA performance and the impact of cryptographic operations, we first evaluate the performance of each cryptographic algorithm. Secondly, we evaluate the latency and bandwidth of sRDMA writes and reads to assess the overheads of secure QPs over insecure QPs. Subsequently, we study the impact of bulk sRDMA operations by measuring the achievable bandwidth for different read/write ratios. Later, we evaluate the performance of the HERD key-value store [17] to examine the impact of sRDMA.

Test settings. The experiments are conducted on two servers directly connected to each other using the RoCEv2 protocol. These servers run Ubuntu 18.04.1 LTS with a 4.15.0-43-generic Linux kernel. Each server is equipped with a Broadcom PS225 25 Gbit/s programmable network controller. Both network adapters have eight-core 64-bit ARM Cortex-A72 3.0 GHz processors and 8 GiB of dual-channel DDR4 DRAM.

6.1 Authentication performance

We first study the performance of the cryptographic engine installed in the SmartNICs. We evaluate 7 different cryptographic algorithms of the openssl 1.1.1a library for message authentication: *aes-128*, *aes-192*, *aes-256*, *chacha20-poly1305*, *sha1-160*, *sha2-256*, *sha2-512*.

Figure 5 depicts the achievable throughput in Gbit/s of those algorithms for different numbers of threads and block sizes. The line rate of the tested RNIC over the RoCEv2 protocol is 20.6 Gbit/s, which is goodput of 25 Gbit/s link. AES algorithms are the fastest for small blocks and achieve 8 Gbit/s for 64 byte blocks using single thread. Thus, our sRDMA library uses the AES128 algorithm as the PRF function needed for key derivation. For larger blocks hash-based methods perform almost as fast as cipher-based algorithms. We observe that chacha20-poly1305 is ~4x slower on average than the AES algorithms. The data also reveals that we cannot achieve the line rate for packet authentication with SHA512.

For varying key sizes of AES algorithms, we have not noticed significant differences in performance and hereafter report results exclusively for the AES128 algorithm. As SHA1-based authentication provides similar performance as SHA256 in all tests, we omit its data in all plots. Additionally, we label chacha20-poly1305 in Figures as *poly1305*.

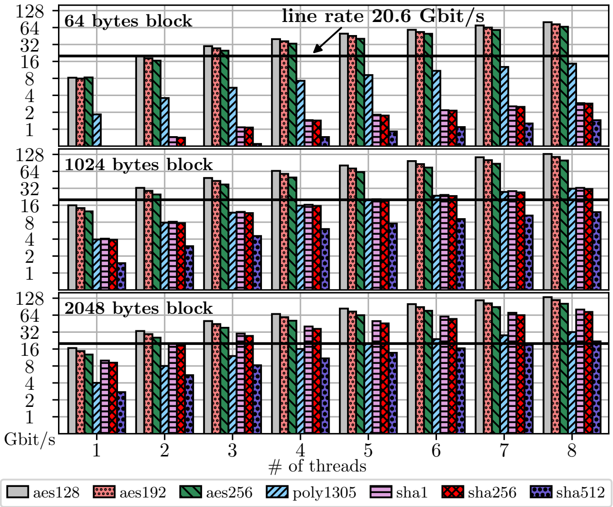


Figure 5: Authentication performance using openssl.

6.2 Evaluation modes

All evaluations have been performed with no security enabled (*NO security*) and in four protection modes:

No security. In *No security* mode RDMA reads and writes are performed as described in Section 5.3 but with skipping packet protection and validation (❷❹❼❾ in Figure 4).

Basic mode. In *basic* mode the key is attached to the secure QP connection directly, so the key is in the RNIC's cache when an incoming packet must be processed.

Pd-prot mode. The secure QP is created without an individual key, but in the secure PD (*pd-prot*) with a key derivation algorithm. We consider the case when the RNIC does not cache derived keys, and therefore, every time a packet arrives, the RNIC must derive the QP key from the PD key. In these experiments we want to show the performance of the system with constant cache misses. Using the cache we expect the same performance as in basic mode without key derivation.

Ext-mem mode. In this mode, the QP is created with an individual key and with extended memory protection (*ext-mem*) enabled. Extended memory protection requires derivation of memory level keys from a PD-level key. In this case, when a packet arrives, the RNIC must generate a key to access memory specified in the RDMA header from the PD-level key and include the generated key in MAC calculation. For this test we also consider that the initiator has the primary memory key which grants access to whole memory region, so the memory key can be derived in one step (depth 0 in Figure 1).

Pd-prot + ext-mem mode. The last mode combines our two protection methods: secure PD and extended memory protection (*pd-prot + ext-mem*). Therefore, the RNIC is responsible for generating both keys when a packet should be processed.

6.3 Latency

To evaluate the overall sRDMA performance and the impact of cryptographic operations, we split latency tests in two cate-

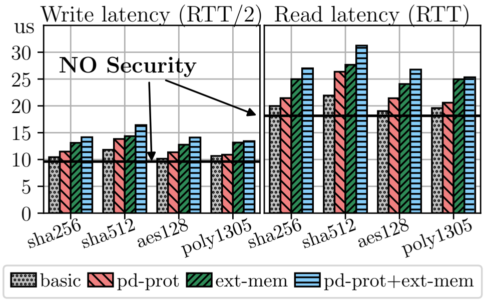


Figure 6: Source authentication latency of reads and writes carrying 32 Bytes payload.

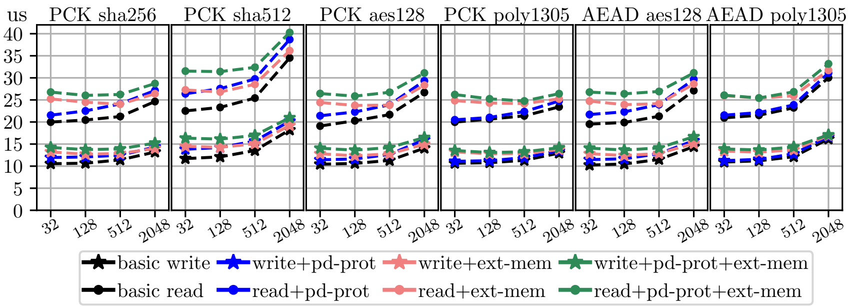


Figure 7: Latency of packet authentication (*PCK*) and encryption (*AEAD*) as a function of payload sizes.

gories: header authentication only and full packet security.

Header authentication. Figure 6 presents the median latency of sRDMA reads and writes in all four protection modes for header authentication. The figure reports the median only as for all measurements deviation from the median is less than 0.4 μs. All measurements are done for packets carrying the payload of 32 bytes. sRDMA write latency is measured for a half round trip, whereas sRDMA reads are for a full round trip. The latency of sRDMA writes without security is 9.55 μs and of sRDMA reads is 18.2 μs which build the baseline for our experiments.

Figure 6 shows that all tested security algorithms in the first mode add about 0.9 μs for sRDMA writes which is approximately 9% more than the insecure version. Another interesting observation is that the QP key derivation is more expensive than memory key derivation. The difference stems from the fact that a key-derivation process involves reinitialization of cryptographic contexts and different algorithms have different reinitialization performance (e.g., AES generates round keys [9]). The same phenomenon occurs for sRDMA reads. As expected, the highest latency is achieved for sRDMA operations with both key derivation and extended memory protection.

Packet security. We evaluate the latency of packet authentication (*PCK*) and packet encryption (*AEAD*) for different payload sizes and in four protection modes. Figure 7 illustrates the median latency of sRDMA reads and writes for SHA256, SHA512, AES128, and POLY1305. In each subplot, the top four lines illustrate sRDMA read round-trip latency, and the bottom four lines half-round-trip latency of sRDMA writes.

Figure 7 highlights that payload authentication is more expensive than header authentication. It takes 15 μs to write and secure 2 KiB payload in the first mode in comparison to header authentication of the same packet with the median of 12 μs. The graph also illustrates that latency increases for both reads and writes with payload size as more data must be authenticated. For AEAD, latency goes up even faster with respect to payload size since more data is en-/decrypted. As anticipated, SHA512 has the highest latency as the most expensive algorithm. We observe that for smaller payload sizes payload authentication and payload encryption achieves approximately the same performance in terms of latency.

6.4 Bandwidth

We measure performance separately for sRDMA reads and writes. As for latency benchmarks, all evaluations are performed in four protection modes. Our implementation is multithreaded where each thread can process requests from a single secure QP. The number of threads represents the number of connections between endpoints. For n threads, each host establishes n secure connections with its SmartNIC, and SmartNICs establish n connections between each other. Thread workers on a SmartNIC do not share any resources and are pinned to distinct cores. In all evaluations the initiator issues requests continuously to the target, but with a limited number of outstanding requests (96 per connection). Once the initiator receives the signal for an sRDMA request completion it posts new requests to maintain 96 outstanding requests. The payload size is 2,048 bytes and bandwidth is measured in Gbit/s of goodput. We also assume the worst case scenario for the secure PD mode (*pd-prot*): the RNIC derives the QP key from the PD key for each packet. In other words, we consider the case when the RNIC does not cache derived keys. The main reason for that is that *pd-prot* mode with caching has the same performance as *basic* mode.

Figure 8 depicts communication bandwidth for sRDMA writes with different cryptographic algorithms. The black line (*NO*) in the header column stands for sRDMA writes with no security enabled. We observe that the single-threaded test with no security achieves only 8 Gbit/s while the highest RDMA goodput bandwidth achievable on our interconnect is 20.6 Gbit/s. The slowdown is caused by processing and parsing headers of messages by the general purpose ARM CPUs of the SmartNICs. Even if no security is enabled, a thread worker reads and parses headers of incoming packets and, depending on the operation code, initiates RDMA requests according to our implementation described in Section 5.3. In our tests we treat performance of secure operations with no security as the baseline. The highest achievable goodput bandwidth with no security is 20.5 Gbit/s which is line rate.

Figure 8 illustrates that sRDMA writes with header authentication can achieve line rate in all four protection modes

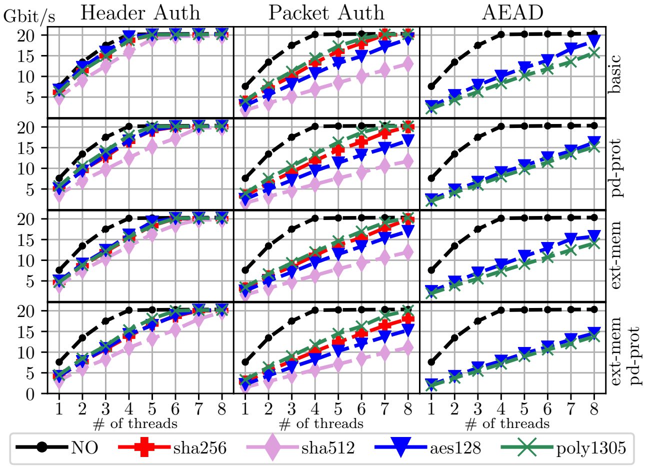


Figure 8: Bandwidth of sRDMA Writes in four different protection modes, and with *NO* security enabled.

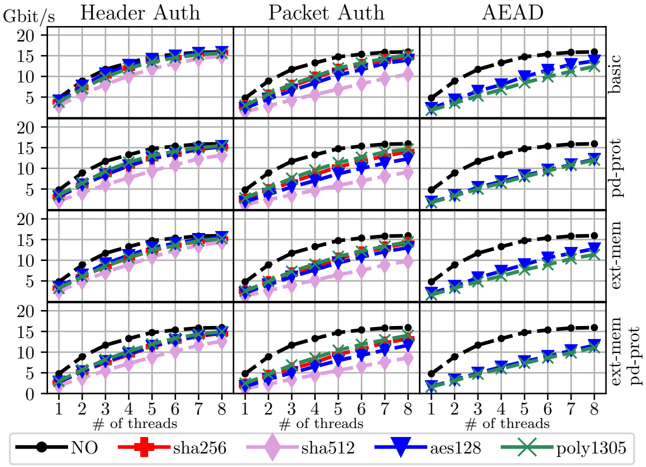


Figure 9: Bandwidth of sRDMA Reads in four different protection modes, and with *NO* security enabled.

if we use all 8 threads. The slowest header authentication is observed for SHA512 due to hashing performance. For full packet authentication SHA512 reaches only a goodput of 13 Gbit/s which is even slower than AEAD algorithms. In the payload encryption mode, our implementation can also achieve line rate for the SHA256 and POLY1305 algorithms. AES128 based authentication achieves 19.6 Gbit/s which is 95% of the line rate. The data also demonstrates that key derivation algorithms slow down sRDMA writes by 2 Gbit/s on average. However, in header authentication mode all algorithms can achieve 20 Gbit/s without performance loss when all 8 threads are used. Another interesting observation is that POLY1305 is faster than AES128 in packet-authentication mode, but slower in packet-encryption one. In AEAD mode, the highest write bandwidth of 19 Gbit/s is observed for the AES128 algorithm.

We have performed a similar benchmark for sRDMA reads in various protection modes. Results of our evaluations are depicted in Figure 9. Again, the black line (*NO*) stands for no security installed and represents the baseline for sRDMA reads. sRDMA reads are more expensive than writes despite the fact that they transfer the same amount of protected bytes as signaled sRDMA writes. Both sRDMA operations require six hops for a full round trip, and they both transfer the same payload size but in different directions. For writes, data is sent from the initiator to the target, and for reads from the target to the initiator. The differences in performance stem from the fact that an sRDMA read is a more complex operation than an sRDMA write and requires to create a special read context and matching it at initiator's SmartNIC (see Section 5.3). In addition, receive buffers on SmartNICs for reads and writes have different lifetimes. For example, a receive buffer can be released on the target SmartNIC once the completion of the RDMA write is received (❻ in Figure 4(a)), however, for reads the buffer on the target SmartNIC can be released once the completion of ❽ is received from Figure 4(b). According to the data, the highest achievable sRDMA read bandwidth is 16.71 GBit/s for 8 threads and about 4.7 Gbit/s for single-threaded test. Overall, our measurements indicate that reads are 16% slower than writes for all tests due to the complexity of sRDMA reads.

CPU Usage in Bandwidth Experiments. In our experiments, sRDMA introduces no overhead on the host CPU usage as packet processing is fully offloaded to the SmartNIC. The host application only needs to submit an RDMA request to the SmartNIC, which performs all cryptographic computations as described in Section 5.3. The SmartNIC, on the other hand, has full CPU usage in almost all experiments, which can be observed in the inability of the majority of security schemes to achieve line rate. The main reason for that is the SmartNIC needs to load the incoming packets from its DRAM to the L1 cache of its CPU cores in order to process the packets depending on the installed security level. Thus, all protection levels which require the CPU to read the whole packet have 800% CPU usage for 8 worker threads, even though in authentication performance experiment (see Figure 5) all authentication algorithms achieves the line rate for 2 KiB blocks. It comes from the fact that the packet authentication and AEAD are memory-bound problems, and, therefore, CPU works at full capacity to copy the data to its caches.

Header authentication requires reading only the header to authenticate the packet. Thus, header authentication algorithms could achieve 100% of line-rate, although, the performance still suffers from cache misses. The lowest CPU usage is observed for AES128 authentication scheme, which is 440% CPU usage for the bandwidth experiment with sRDMA Write requests. The sRDMA reads, on other hand, consume almost 750% on the target SmartNIC.

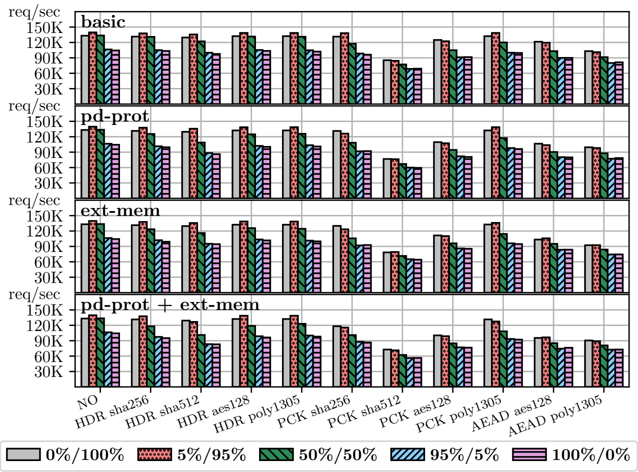


Figure 10: Throughput of mixed read/write benchmark.

6.5 Mixed write/read workload

The results of Figure 8 and Figure 9 are valid for either read-only or write-only workloads, which are uncommon for read-world applications. Therefore, we measure the throughput of sRDMA in a more realistic scenario as used in key-value stores that exploit one-sided RDMA operations. Figure 10 shows the throughput for workloads with different (read/write) ratios, including write only (0%/100%), write mostly (5%/95%), equal-shares (50%/50%), read-mostly (95%/5%) and read-only (100%/0%). The read-heavy workload is representative for applications such as photo tagging. The update-heavy workload is typical for applications such as an advertisement log that records recent user activities. In this benchmark the payload size is 2,048 bytes, and sRDMA is deployed with all 8 workers. We also consider the worst case scenario for the secure PD mode (*pd-prot*), when the RNIC derives the QP key for each packet. The *pd-prot* mode with QP key caching has the same performance as *basic* mode.

Figure 10 illustrates that (5%/95%) workload performs better than (0%/100%) one. The reason for that is better utilization of the bi-directional connection between endpoints since sRDMA writes send data from the initiator to the target, whereas sRDMA reads from the target to the initiator. Therefore, in that case we achieve a better utilization of the connection in the direction of the initiator. In theory, a (50%/50%) ratio would lead to the highest throughput as both links would be loaded evenly; however the lower performance of sRDMA reads overwhelms benefits of the network utilization. For the same reason, the throughput decreases for higher read ratios.

6.6 Key-value store workload

HERD [17] is an RDMA-accelerated key-value store which uses a mix of RDMA write and IB send verbs. HERD uses MICA's [25] algorithm for both GETs and PUTs: each GET requires up to two random memory lookups, and each PUT requires one. In HERD, clients transmit their request to the server's memory using RDMA writes, and get responses via unreliable datagram QPs. To comply with our sRDMA design, we made some changes to the original HERD implementation. First of all, we replace all unreliable datagram QPs with RC QPs as they are not reliable and not point-to-point and thus incompatible with sRDMA. That is, the server replies to clients via RC QPs, but still uses IB Send verbs. For that, we also implement secure SEND operations which are similar to sRDMA writes, but they always generate the completion on the target and do not require knowing destination buffers. Since an IB Send request does not contain the header with destination buffer, it does not support extended memory protection. The second change is that clients send requests via reliable sRDMA writes instead of unreliable writes.

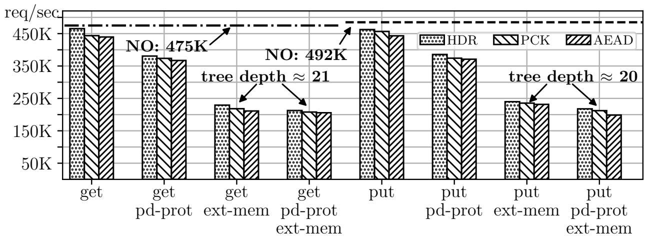


Figure 11: Throughput of the HERD kvs over sRDMA.

Key-value-store (KVS) experiments use one server machine and one client machine. The server machine has only one worker process when the client machine has 8 processes. Each client process establishes an sRDMA connection to the server. The key size is 16 bytes and the value size is 32 bytes. Therefore, clients send and receive small messages of less than 40 bytes. The KVS contains 8,388,608 keys and occupies 1 GiB of memory. Figure 11 depicts the throughput for puts and gets in different protections modes based on the AES128 cipher. We also measure HERD's throughput with NO protection which is 475K req/sec for gets and 492K req/sec for puts. Puts are faster than gets because they cause fewer lookups in internal memory structures.

According to the data in Figure 11, basic packet authentication without key-derivation algorithms achieves almost the same throughput as the unprotected version. Interestingly, even the AEAD mode decreases the throughput by 7.3%. In the setting with a secure PD when the key must be generated for each request, we observe a 21% slow down in both puts and gets. It is worth mentioning that we intentionally derive the QP keys for each request in the secure PD mode (*pd-prot*) to see the effect of constant misses in QP keys. In real settings, an RNIC would have a cache with generated keys to reduce computation. In such case, the *pd-prot* mode has the same performance as *basic* mode.

A drastic decrease in performance can be observed for evaluations with enabled extended memory protection. The reason for this is that HERD's clients WRITE their GET

Table 4: Comp. of sRDMA to IPSec and TLS over RoCE.

Protocol	Sec. comm.	IBA supp.	One-sided comm.	Hdr overhead.
RDMA	✗	✓	✓	-
IPSec	✓	✗	✗	50-80 B
(d)TLS	✓	✗	✗	25-40 B
[23, 24]	✗	✗	✓	12-16 B
sRDMA	✓	✓	✓	12-64 B

requests of 17 bytes and PUT requests of 40 bytes to the contiguous memory region of 16 MiB on the server machine. Therefore, it takes on average 20 steps for PUTs and 21 steps for GETs to derive the memory MAC using our binary tree approach, which causes such significant drop in performance. To alleviate the problem, the depth of the tree can be limited to 0, and then the *ext-mem* would achieve the same performance as the *pd-prot* case.

7 Related Work on Securing IBA

RFC 5042 [32] analyzes the security issues around uses of RDMA protocols. It reviews various attacks against resources including spoofing, tampering, information disclosure, and DoS. As a countermeasure the authors suggest to employ IPsec authentication and encryption [10]. However, IPSec currently does not support RDMA traffic, because it is unaware of the encapsulated RDMA headers and thus cannot distinguish QP endpoints. A naive application of IPSec to RoCE packets would not achieve source authentication as all RoCE traffic is destined to the same UDP port (and not the QPN). Thus, the use of IPsec would incur changes in the packet format, whereas sRDMA is supported by native IBA and RoCE. Additionally, the complexity of IPsec and its high processing overheads [31] make it ill-suited for high-performance and low-latency applications and would introduce a header overhead of 50-80 bytes [21]. While the IPsec-enabled Cavium LiquidIO II [2] and Mellanox Innova [3] NICs support RoCE, they do not support IPsec-based protection of RoCE packets.

Lee et al. [23,24] discuss security vulnerabilities in IBA and show that they could be exploited by an adversary with moderate overhead. The authors suggest to replace the Invariant CRC field with a MAC to achieve packet authentication. Unfortunately, this might lead to routers dropping packets with invalid ICRC, making the proposed solution incompatible with legacy routers. Additionally, they discuss how IBA could reduce its key exposure risk by introducing partition- and queue-level key distributions. However, modifying partition-level keys can lead to packets being dropped as they might be used by routers and switches to enforce partitioning. Furthermore, their design uses the 24 bit PSN as a nonce which cause a reuse of a PSN after 80 ms on modern RNICs [28]. Finally, the authors provide no implementation of their system, but rather simulate the performance of symmetric ciphers to show that they are suitable for high performance networking.

RDMA Side-Channel Attack. Kurth et al. [22] have shown that the Intel DDIO [1] and RDMA features facilitate a side-channel attack named NetCAT. Intel DDIO technology allows RDMA reads and writes access not only the pinned memory region but also parts of the last level cache of the CPU. NetCAT remotely measures cache activity caused by a victim SSH connection to perform a keystroke timing analysis. An attacker can make use of the attack to recover words typed by a victim client in the SSH session from another computer.

Tsai et al. [41] implemented a set of RDMA-based remote sidechannel attacks that allow an attacker on one client machine to learn how victims on other client machines access data. They further extend their work by building side-channel attacks on Crail [38].

Using sRDMA a large attack surface could be removed by permitting only trusted entities to initiate RDMA requests.

8 Conclusion

Using NIC-based authentication and encryption enables secure communication for systems that require high performance guarantees such as RDMA mechanisms. sRDMA provides strong authenticity and secrecy, and prevents several forms of DoS attacks. Thus, safety- and security-critical applications that rely on RDMA must use sRDMA to prevent attacks by malicious entities within the same network.

Our software implementation on the SmartNIC causes a high load due to data movement overheads. The datapath could be optimized with a different architecture using specialized programmable packet processing units [13,20]. Furthermore, sRDMA could also be hardened into fixed logic as the area and power consumption overhead are marginal compared to regular input/output processing [12,16,27]. Additionally, sRDMA minimizes memory consumption on the RNIC using PD-level protection.

Acknowledgment

We thank our shepherd, Heming Cui, and the anonymous reviewers for their helpful feedback. We thank Broadcom Inc., especially Fazil Osman, for the donation of two SmartNICs as well as continuous support. We gratefully acknowledge support from ETH Zurich, and from the Zurich Information Security and Privacy Center (ZISC). Furthermore, we thank for Microsoft Swiss Joint Research Centre for support.

References

[1] Intel® Data Direct I/O Technology Overview. https://www.intel.co.jp/content/dam/

www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2019. [Accessed 15-May-2020].

[2] LiquidIO®II 10/25G Smart NIC Family. https://www.marvell.com/documents/08icqisgkbtn6kstgzh4/, 2019. [Accessed 15-May-2020].

[3] Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf, 2019. [Accessed 15-May-2020].

[4] InfiniBand Trade Association et al. The InfiniBand architecture specification. 2000.

[5] Infiniband Trade Association et al. Supplement to InfiniBand Architecture Specification Volume 1, Release 1.2. annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[6] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

[7] Broadcom. Stingray 2x25Gb High-Performance Data Center Smart NIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2019. [Accessed 15-May-2020].

[8] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[9] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. 1999.

[10] Naganand Doraswamy and Dan Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 401–414, 2014.

[12] Kris Gaj and Pawel Chodowiec. FPGA and ASIC implementations of AES. In *Cryptographic engineering*, pages 235–294. Springer, 2009.

[13] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E Grant, and Ron Brightwell. spin: High-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2017.

[14] Russell Impagliazzo, Leonid A Levin, and Michael Luby. Pseudo-random generation from one-way functions. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 12–24. ACM, 1989.

[15] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-17, Internet Engineering Task Force, December 2018. Work in Progress.

[16] Hyun-Wook Jin, Pavan Balaji, Chuck Yoo, Jin-Young Choi, and Dhabaleswar K Panda. Exploiting nic architectural support for enhancing ip-based protocols on high-performance networks. *Journal of Parallel and Distributed Computing*, 65(11):1348–1365, 2005.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of ACM SIGCOMM*, pages 295–306, 2014.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the USENIX Annual Technical Conference*, ATC, pages 437–450, 2016.

[19] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[20] Antoine Kaufmann, Simon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[21] Stephen Kent. IP authentication header. Technical report, 2005.

[22] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *S&P*, May 2020. Intel Bounty Reward.

[23] Manhee Lee and Eun Jung Kim. A comprehensive framework for enhancing security in InfiniBand architecture. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

[24] Manhee Lee, Eun Jung Kim, and Mazin Yousif. Security enhancement in InfiniBand architecture. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 10–pp. IEEE, 2005.

[25] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 429–444, 2014.

[26] Chu-Hsing Lin. Dynamic key management schemes for access control in a hierarchy. *Computer communications*, 20(15):1381–1385, 1997.

[27] Bin Liu and Bevan M Baas. Parallel AES encryption engines for many-core processor arrays. *IEEE transactions on computers*, 62(3):536–547, 2013.

[28] Mellanox. ConnectX-6 EN Single/Dual-Port Adapter. https://www.mellanox.com/products/infiniband-adapters/connectx-6, 2019. [Accessed 15-May-2020].

[29] B Clifford Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of IEEE International Conference on Distributed Computing Systems-ICDCS*, pages 283–291. IEEE, 1993.

[30] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[31] Jungho Park, Wookeun Jung, Gangwon Jo, Ilkoo Lee, and Jaejin Lee. Pipsea: A practical ipsec gateway on embedded apus. In *Proceedings of ACM Conference on Computer and Communications Security*, CCS, pages 1255–1267, 2016.

[32] J. Pinkerton and E. Deleganes. Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security. RFC 5042, October 2007.

[33] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. Technical report, 2007.

[34] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012.

[35] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[36] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[37] Phillip Rogaway. Nonce-based symmetric encryption. In *International Workshop on Fast Software Encryption*, pages 348–358. Springer, 2004.

[38] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[39] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proceedings of EuroSys Conference*, EuroSys, pages 39:1–39:14, 2018.

[40] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2019. [Accessed 15-May-2020].

[41] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. Pythia: remote oracles for the masses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 693–710, 2019.

[42] VMWare. ESXi VM and Hypervisor Escape Advisory. https://blogs.vmware.com/security/2018/11/vmware-and-the-geekpwn2018-event.html, 2019. [Accessed 15-May-2020].

UREQA: Leveraging Operation-Aware Error Rates for Effective Quantum Circuit Mapping on NISQ-Era Quantum Computers

Tirthak Patel, Baolin Li, Rohan Basu Roy, and Devesh Tiwari
Northeastern University

Abstract

Noisy Intermediate-Scale Quantum (NISQ) computers are enabling development and evaluation of real quantum algorithms, but due to their highly erroneous nature, careful selection of qubits to map the algorithm on to real hardware is required to minimize the error rate of the algorithm output. In this paper, we propose UREQA, a new approach that introduces quantum-operation-aware error rate prediction to minimize of output errors of quantum algorithms running on NISQ devices.

1 Introduction

Noisy Intermediate-Scale Quantum (NISQ) computers are enabling development and evaluation of quantum algorithms in various domains including chemistry and physics simulations, combinatorial and black-box optimization, and quantum cryptography [16, 20]. A major challenge toward increasing the practicality and wide-adoption of quantum computing is the high error rate observed on current NISQ devices, often orders of magnitude higher than the classical computing systems [16,20]. This challenge is likely to remain prevalent in the near-term, and requires innovative techniques to mitigate the side-effects of quantum errors [6, 12, 16, 20]. Next, we provide a brief background of quantum computers, errors, and algorithm execution.

Background and Problem Statement. In quantum computers, a *qubit* is the fundamental unit, analogous to a classical bit. A qubit state ($|\Psi\rangle$) can be expressed as a superposition of the two basis states: $|0\rangle$ and $|1\rangle$. More formally, $|\Psi\rangle = a|0\rangle + b|1\rangle$, where a and b are complex numbers such that $||a||^2 + ||b||^2 = 1$. Due to the quantum physical behavior of qubits, upon measurement, this superposition collapses, and the qubit is found either in state $|0\rangle$ (with probability $||a||^2$) or in state $|1\rangle$ (with probability $||b||^2$). Multiple operations can be sequentially performed on a set of qubits on a computer to run a quantum algorithm. Upon the completion of the execution of a quantum algorithm, the qubit state is measured for all qubits and the output is analyzed.

The qubit operations, referred to as *quantum operations*, are of three types: 1-qubit gates, 2-qubit gates and readout. 1-qubit gates operates on a single qubit and change the superposition state of the qubit. 2-qubit gates entangle two qubits and can change the state of the "target" qubit depending on the state of the "control" qubit (Sec. 2 provides more details on these operations). The readout operation simply refers to measuring a qubit's state. The readout operation is applied only at the end of the execution as it destroys the qubit's state. A *quantum algorithm* is expressed as a sequence of gate operations operating on multiple qubits. The mapping of a logical quantum algorithm on the physical qubits of a quantum computer is referred to as a *quantum circuit* — a set of quantum operations and the corresponding qubits on which the operation is being performed. An example of the execution of a quantum circuit is shown in Fig. 1. This example circuit has three qubits all initialized to the ground state $|0\rangle$. A few quantum gates are applied to them (details provided in Sec. 2) and their states are measured at the end and transferred over to a classical computer. It is important to note that a *single* quantum algorithm can map to *multiple* circuits in different ways on the same computer, much like how a classical algorithm can be executed on any permutation of transistors.

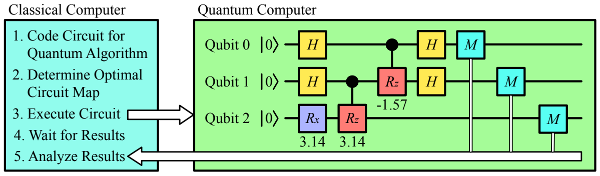


Figure 1: Execution flow of Quantum Phase Estimation (QPE) algorithm mapped to a quantum circuit. Each horizontal line represents a qubit, and each box represents a quantum operation. The time order of operations flows from left to right.

Unfortunately, qubits and operations on NISQ machines are highly error-prone and hence, quantum circuits have erroneous output. State-of-the-art approaches carefully choose the qubits and operations with the lowest overall error rate for determining the best circuit map [3, 15, 18, 25–28, 31, 33]. Error rate of each operation is estimated based on the historical information (e.g., IBM calibrates and publishes error rate for all qubits of a machine on a twice-a-day basis). Based on these error rates, a "good" circuit map is chosen to obtain an outcome that has a high probability of being close to the correct output. Estimating error rate correctly is the key to choosing the optimal circuit map for a given quantum algorithm. For example, if the estimated error rate of each qubit is significantly different than the actual error rate when the circuit map is executed, then these differences add up over the circuit map execution and result in a significantly inaccurate outcome. *Therefore, previous works have focused on estimating the error rates accurately and using that to find the optimal circuit map [3, 15, 18, 25, 26, 28].*

What is Missing from Existing Solutions? Current approaches use a single number to characterize the error rate of a given qubit irrespective of the different quantum operations being performed on the qubit [25, 26, 28, 31]. For the first time, we show that error rate is not only qubit-specific, but also operation-specific (as explained in Sec. 2, a 1-qubit gate can perform different types of quantum operations on a single qubit). We show that quantum error rates can vary significantly depending on the specific quantum operation that is being performed, even if other conditions are kept constant (i.e., the physical qubit and the machine). Some qubits with low aggregate average error rate might experience high error rate for specific quantum operations. Hence, these qubits should be avoided for a circuit-map selection if a particular circuit consists of many such specific quantum operations. The reason for this phenomena is the unstable nature of current NISQ technology where qubits are erroneous and do not have consistent properties as different qubits interact differently with external control and the environmental features. *This is the first work to discover and leverage the above insight to choose better circuit maps that lower the impact of quantum errors, and push the state-of-the-art in improving the efficiency of quantum algorithm execution on NISQ computers.*

UREQA Solution. UREQA[1] builds a data driven model for correctly estimating the error rates of operations on different qubits of a quantum computer, and then, leverages this information to find the most optimized circuit for a quantum algorithm. UREQA builds its error rate prediction model by performing a large number of experiments on real IBM NISQ computers. Our evaluation shows that these error rate prediction methods are more accurate than current state-of-the-art approaches of simply using a general error rate number periodically published by the quantum computing platform provider.

To demonstrate UREQA's effectiveness, we evaluate UREQA for a diverse set of quantum benchmarks, conducting experiments over more than 50 days on four different quantum computers in the IBM QX cloud. Our results show that our operation-aware solution achieves a small median prediction error rate of 1%. Using these operation error-rate prediction models, UREQA's optimized circuit map selection achieves up to 15% reduction in error rate for a quantum algorithm, compared to the current approaches which rely on a single aggregated number for error rate estimation based on historical data.

Table 1: IBM QX quantum computers.

Online Date	Computers (Num. Qubits)
Nov 06, 2018	Melbourne (14), Yorktown (5)
Jul 03, 2019	Ourense (5), Vigo (5)

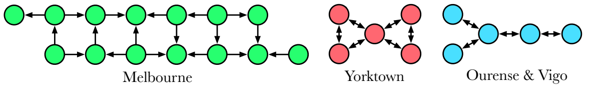


Figure 2: Layout of IBM quantum computers. The circles represent qubits. The arrows show possible 2-qubit gates: the direction points from control to target qubit.

UREQA's quantum error prediction model and circuit mapping framework is open-sourced at `https://github.com/GoodwillComputingLab/UREQA`.

2 UREQA: The Solution

Background. This study is performed on the IBM Quantum Experience (QX) - a public cloud service. We use the IBM QX machines listed in Table 1. They cover a diverse range of quantum architectures in terms of error-rates, topology, and time of introduction (Fig. 2).

Quantum operations on these computers include both the gate and readout operations. Primary 1-qubit gates include the Hadamard (H) gate which puts the two basis states into equal superposition and the x-, y-, and z-rotation gates (R_x, R_y, and R_z, respectively) which rotate the qubit about the x-axis, y-axis and z-axis on the Bloch Sphere, respectively. The Bloch Sphere is a unit sphere with the $|0\rangle$ state represented as a vector pointing toward the positive z-axis and the $|1\rangle$ state is represented on the negative z-axis. The other two axis represent the qubit phase. The qubit state vector can point anywhere on the Bloch Sphere, but upon readout, it collapses to the positive ($|0\rangle$) or negative ($|1\rangle$) z-axis. As an example, Fig. 3 uses the Bloch Sphere to show the state changes after applying a H gate followed by a R_z gate with π rotation to a single qubit. When the H gate is applied, the qubit state vector points toward the positive x-axis and the qubit is equally probable to be measured as $|0\rangle$ or $|1\rangle$. This probability of measurement remains the same even after a R_z rotation is applied, except the qubit has a negative phase.

All 1-qubit gates have 2-qubit variants (CH, CR_x, CR_y and CR_z) where one qubit is the control and the other is the target. The respective 1-qubit gate is applied to the target qubit depending on the superposition of the control qubit. In Fig. 1, the connection between qubit 0 and R_z gate of qubit 1 means it is a CR_z gate with qubit 0 as control and qubit 1 as target.

These qubit operations can be erroneous. IBM's qubits are fixed-frequency superconducting Transmon qubits

[1] UREQA (Eureka) stands for utilizing operation-aware error rate predictions (for better circuit mapping) on quantum computers.

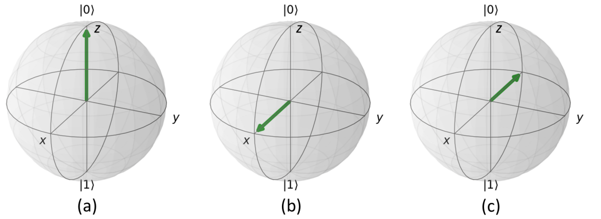


Figure 3: A qubit (green arrow tip) on a Bloch sphere. The qubit in (a) first gets manipulated by an H gate to state in (b), then by a R_z gate to state in (c).

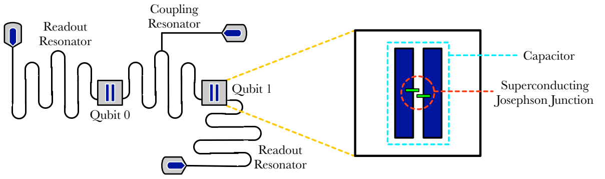


Figure 4: Design of IBM's superconducting qubits technology.

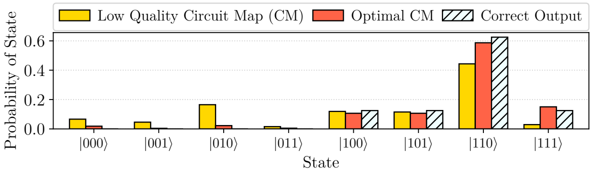


Figure 5: Choice of circuit map can greatly impact overall output error: different circuit maps for the QPE algorithm.

based on Josephson Junctions, and the Transmon frequency is referred to as the **qubit frequency**. On IBM's quantum computers, the qubits are implemented using Josephson Junctions created by separated superconducting electrodes and capacitors as shown in Fig. 4. 1-qubit gates are performed by applying external controls in the form of microwave pulses. Errors in applying these pulses cause **1-qubit gate errors**. Entanglement between two qubits is performed using coupling resonators. These coupling resonators can be highly erroneous causing **2-qubit gate errors**. Lastly, readout operation (or qubit state measurement) is performed using readout resonators as shown in Fig. 4. The readout resonators are also highly error-prone and cause **readout errors** when qubit states are measured. In fact, other factors can also affect error rates. Once initialized, a qubit can only retain its state for a limited time (coherence time). There are two types of coherence times: (1) The **T1 coherence time** is associated with amplitude damping due to the qubit's natural energy decay to the ground state. (2) The **T2 coherence time** is associated with phase damping due to environmental factors.

IBM's computers are calibrated twice a day, and the qubit coherence times change after each calibration. We note that the error rates are determined when calibration tasks are performed for all the operations of a quantum computer. Calibration is the task of determining qubit frequency and accordingly, setting the properties of the microwave tone which changes the state of a qubit. During calibration, operation characteristics such as the frequency of a qubit and the optimal microwave tone amplitude are determined based on new properties of the qubit. These characteristics are then used to perform all the operations. These new characteristics determine the error rate of the operation. The effect of environmental factors (such as the electromagnetic interference, fluctuating temperature, or mechanical vibrations) is already captured in the operation error rates. Coherence times are also measured immediately after calibration is performed. Note that regular circuits (jobs) cannot run on machines when calibration is being performed. Thus, it is impractical to constantly keep calibrating the machines, and hence, this practical constraint forces the calibration to be performed typically twice daily.

Current Efforts in Circuit Mapping. IBM posts a single error number for all 1-qubit and 2-qubit gates for each qubit twice a day. One solution to the aforementioned circuit mapping problem can be to map quantum operations on qubits which have the minimum operation error rates according to these posted numbers [10, 25, 26, 28]. The idea is to maximize the Estimated Success Probability (*ESP*) of a quantum circuit [25]. The ESP is calculated as $\prod_{i=1}^{N_{gates}} g_i * \prod_{j=1}^{N_{readout}} m_j$, where g is the success rate of gates and m is the success rate of readout (success rate = 1 - error rate). The circuit map with highest ESP is the optimal circuit map. Fig. 5 shows the impact of choosing a low quality circuit map vs. an optimal circuit map for executing the quantum phase estimation (QPE) algorithm. The correct output of QPE has states $|100\rangle$, $|101\rangle$, and $|111\rangle$ with probability 0.125, and state $|110\rangle$ with probability 0.625. On real-systems, executing a circuit map results in state probabilities that are different than the correct probabilities. Using the correct probabilities as reference, the optimal circuit map has an overall error of 6% (sum of errors of all states divided by 2), while the low quality circuit has an overall error of 28%. Thus, estimating the ESP of a circuit map accurately (and hence, in turn estimating the error rate of quantum operations) is critical for mitigating the side-effects of erroneous quantum operations. However, current approach of using the published numbers to estimate the error rates implicitly assumes that all 1-qubit operations have uniform errors and that all 2-qubit operations also have uniform errors. This is far from the actual behavior of errors as we show next.

Different quantum operations exhibit significant variation in observed error rates. Fig.6 shows that quantum errors are correlated with the specific type of operation being performed (on the same qubit; results are averaged over all available qubits and platforms for simplicity). For example, on IBM computers R_z is imple-

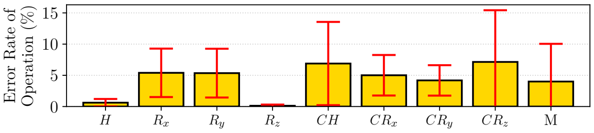


Figure 6: Different quantum operations can have significant different error rates with a high degree of variance.

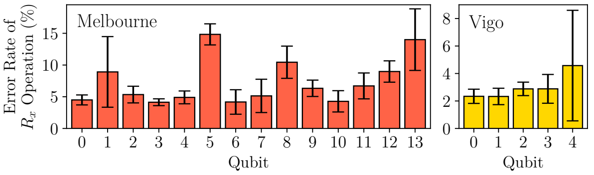


Figure 7: Error rate of a quantum operation varies across machines and among qubits within the same machine.

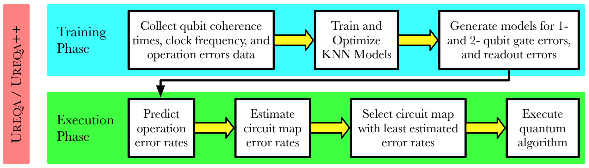

UREQA / UREQA++
Training Phase
Collect qubit coherence times, clock frequency, and operation errors data
Train and Optimize KNN Models
Generate models for 1- and 2- qubit gate errors, and readout errors
Execution Phase
Predict operation error rates
Estimate circuit map error rates
Select circuit map with least estimated error rates
Execute quantum algorithm

Figure 8: Key steps in UREQAworkflow.

Table 2: Predictive features of different operations.

Operation	**Predictive Features**
1-Qubit Gate	Computer ID, Qubit T1 Coherence Time, Qubit T2 Coherence Time, Qubit Frequency, Gate Type (H, R_x, R_y, R_z)
2-Qubit Gate	Computer ID, Control Qubit T1 Coherence Time, Control Qubit T2 Coherence Time, Control Qubit Frequency, Target Qubit T1 Coherence Time, Target Qubit T2 Coherence Time, Target Qubit Frequency, Gate Type (CH, CR_x, CR_y, CR_z)
Measurement (Readout)	Computer ID, Qubit T1 Coherence Time, Qubit T2 Coherence Time, Qubit Frequency

mented as a simple frame change with no physical computation; hence, it has close to 0 error rate. The H gate also has a low error rate; however, the error rates of the other two 1-qubit gates R_x and R_y are much higher. 2-qubit gates like CH and CR_z also have high error rates. It is a conventional belief that 2-qubit gates have a higher error rate than 1-qubit gates [10, 23, 27]. However, our analysis reveals that while on average 2-qubit gates have higher error rates than 1-qubit gates, certain types of 1-qubit gates such as R_x and R_y have error rates comparable to 2-qubit error rates and readout (M) error rate.

A potential reason for varying error rates among different quantum operations can be the difference in microwave tones that are applied to implement the operation on a gate. For example, an R_x gate with a π rotation (on a single qubit) is applied using a Gaussian microwave pulse of a certain calibrated amplitude A. On the other hand, the H gate (also, on a single qubit) is implemented using a Gaussian microwave pulse of half the amplitude ($A/2$) and pre- and post- pulse frame changes. It can lead to a lower error rate because its pulse has half the amplitude of the R_x gate and the frame changes have zero error. Overall, this finding of operation-specific error rates motivates the need for accounting for the operation type when estimating the error rates.

Errors rates also vary temporally and spatially across computers and qubits. In Fig. 6, the error bars show the high standard deviation (variation) of operation error rates across time. This situation is further exacerbated by the fact that error rates for the same operation also vary across different computers and qubits, as shown in Fig. 7 for the R_x gate. The variation in the error rates for R_x across different qubits is considerable and has not decreased even in the newest IBM quantum computer, Vigo. The error rates vary across qubits and across operations performed on the same qubit.

UREQA Overview. While using published errors does offer simplicity, accurate estimation of ESP needs better prediction of operation error rates. The high degree of instability and uncertainty makes it difficult to model the behavior of these errors using analytical or rule-based models. Therefore, UREQA takes a data-driven machine-learning-based approach, as shown in Fig. 8. We develop a data-driven model which helps perform accurate predictions of error rates of individual operations. Then, when executing quantum algorithms, these pre-trained models can be used to estimate circuit map error rates and the best circuit map can thus be selected to execute the algorithm with minimal errors.

UREQA Model Development. To collect the error rate data for different types of operations, we developed micro benchmarks that perform a specific operation on every qubit of all available quantum computers. For example, to get the readout error, every qubit was measured in its initialization state of $|0\rangle$ without running any gate operation. To get the error of H, R_x, R_y, R_z, CH, CR_x, CR_y, and CR_z, the corresponding gate-operation was performed, and result was measured and compared against the ground truth. Our automated workflow collected over 20,000 samples. Each run consisted of 1024 trials — multiple trials need to be conducted because the output of a quantum circuit is probabilistic. Other data such as coherence times and frequencies of individual qubits were obtained from IBM's daily calibration results.

The goal of model development is to predict the error rate of a given operation given a set of predictive features. A complete list of these features is provided in Table 2 for the three types of quantum operations. For example, given qubit 0 on Melbourne computer's T1 coherence time, T2 coherence time, and frequency after today's calibration, predict its readout error rate. The predictive features are chosen based on their availability on the IBM QX machines and based on their relevance to the qubit operation (e.g., T1 coherence time of qubit 1 is relevant to the 1-qubit error rate of the same qubit but not to other qubits on the machine). Principle Component Analysis (PCA) was performed to determine the features

Table 3: Optimal parameters tuned for KNN learners.

Operation	Number of Neighbors	Distance Metric	Distance Weight
1-Qubit Gate	13	Euclidean	Squared Inverse
2-Qubit Gate	31	Correlation	Inverse
Readout	68	Jaccard	Inverse

which contribute the most to the variance of the dataset and the features shown in Table 2 were found to account for more than 95% of all variance in the dataset.

After thorough experimentation and hyper-parameter tuning, we assessed that k Nearest Neighbors (k-NN) classification learner [1] was the best learning model for error rate prediction – it has the lowest mis-classification error. The output of a k-NN learner is a membership to a class (class here refers to a particular error rate). A sample is classified by a plurality vote of its neighbors given a set of predictors. The sample gets assigned the class most common among its k nearest neighbors.

The hyper-parameter optimization of the k-NN learners was performed using Bayesian Optimization [21] which builds a stochastic model of the parameter space by progressively sampling parameters which have the highest expected improvement based on the constructed model. The parameters optimized include number of nearest neighbors involved in the voting process, distance metric (e.g., Euclidean, Manhattan, etc.), and distance weight (e.g., Equal, Inverse, etc.). Table 3 shows the optimal parameters obtained after performing Bayesian-Optimization-based hyper-parameter tuning for the three error rate learners. 90% of the dataset was used for training and 10% was used for testing. Data was randomized so as to ensure that all quantum computers, qubits, and operation characteristics were included in the training dataset. The training was performed using 5-fold cross validation [32] to avoid machine-learning methodology pitfalls such as over-fitting the model.

Finally, we note that UREQA is practically feasible. The model building and training can be completed within a few days. This process needs to be invoked or refreshed when significant architectural/operational changes are introduced to a particular machine. UREQA's result quality is not highly sensitive to the number of samples (e.g., an incremental improvement in result quality over 10,000 samples is limited). Note that the investment of one-time 20,000 sample based training can potentially be amortized over multiple months in choosing better circuit mappings which reduce the error rate.

3 UREQA: Evaluation and Analysis

Quantum Algorithms and Circuits. Real quantum benchmark algorithms were programmed using *Qiskit*, IBM's python-based language for quantum circuits [2], and executed on different IBM computers (benchmarks are listed in Table 4). The results from these circuits are used to assess the improvement in circuit mapping due to improved prediction power. *Note that the training data of individual operation execution and the assessment data of execution of real quantum circuits were generated at disjoint sets of time periods to avoid biasing the results.*

Table 4: Quantum benchmarks used for evaluation.

Benchmark ID	Benchmark description
$BV2$	2-Qubit Bernstein-Vazirani [5]
$BV3$	3-Qubit Bernstein-Vazirani [5]
QPE	Quantum Phase Estimation [8]
SIA	Simon's algorithm [13]
HR_x^8H	Circuit to stress X gate errors (expected output $\|0\rangle$)
$R_xHR_x^8H$	Circuit to stress X gate errors (expected output $\|1\rangle$)

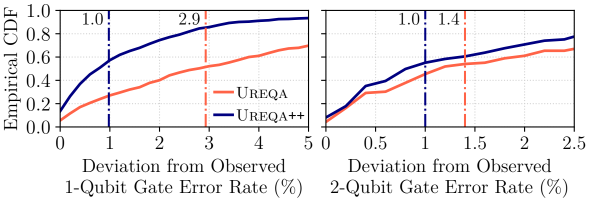


Figure 9: The prediction quality is much better when operation-aware predictor is used in UREQA++.

The benchmarks are chosen to cover a diverse set of quantum algorithm characteristics. For example, the Bernstein-Vazirani (BV) benchmarks heavily use H and R_z gates and the QPE benchmark has high number of 2-qubit gates such as the CR_z gates. Finally, two home-grown benchmarks HR_x^8H (H gate followed by 8 R_x gates followed by H gate) and $R_xHR_x^8H$ are used. These benchmarks are designed to generate and test large 1-qubit gate errors with a long sequence of R_x gates.

Evaluation Metrics. The prediction quality of a model is assessed using *deviation of the predicted value from the observed value of the operation error rate*. The effectiveness of a method's prediction on real quantum algorithms is assessed using the *overall output error rate when a circuit map is selected for an algorithm using the prediction provided by a model/method.*

Evaluated Techniques. (1) The base method: Best circuit map is selected by maximizing the ESP using the operation errors posted by IBM as used in current approaches [10,25,26,28]. (2) UREQA: Best circuit map is selected by predicting operation errors using k-NN models trained without using operation-specific information. (3) UREQA++: Best circuit map is selected by predicting operation errors using k-NN models trained with all features including operation-specific information.

Including operation-specific information in the prediction improves the prediction quality significantly. Fig. 9 shows that when operation-specific information (type of 1-qubit or 2-qubit gate) is included as the predictor, the prediction error is much lower on average. For 1-qubit gates, the median deviation from observed error rate is only 1% with UREQA++ compared to 2.9% with

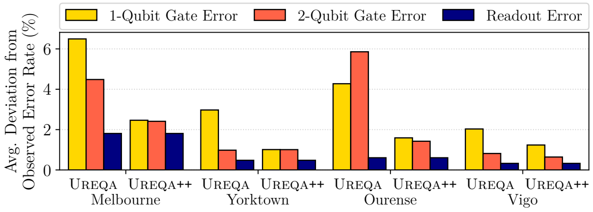


Figure 10: The prediction quality is good across different computers with UREQA++.

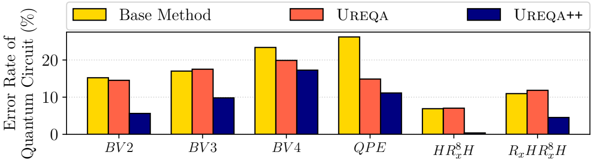


Figure 11: Circuit maps selected using UREQA++ perform much better as compared to other methods.

UREQA. We note that the CDF of UREQA++ has a much steeper rise, indicating that the prediction error is smaller for a large majority of test samples. Similarly, for 2-qubit gates, the median deviation from observed error rate is only 1% with UREQA++ and it is 1.4% with UREQA. As our results show later, even this seemingly small difference has a compounded impact when these predictions are used to estimate the error rate of an entire circuit because multiple operations are used in real quantum algorithms. Note that both UREQA and UREQA++ are hyperparameter optimized using the Bayesian-Optimization-based procedure described earlier. Thus, both models are optimal for their given set of predictor features. Yet, UREQA++ performs better as it is trained in an operation-aware manner.

The deviation from observed value is small across different quantum computers when using UREQA and UREQA++. Fig. 10 shows that UREQA and UREQA++ have another desirable result: the deviation from the observed value is low across the four computers for all the three types of operation errors. Melbourne has slightly higher deviation from the observed error rates than other computers because it has older and more unstable technology which makes error rates vary considerably. This makes prediction difficult. For the other three computers, the average deviation is less than 2% with UREQA++.

Quantum circuit error rate drops significantly when the circuit map is chosen using predictions provided by UREQA++. Fig. 11 shows error rates when the best circuit maps to execute a quantum algorithm are selected using different methods. UREQA achieves similar or significantly better results in some cases (e.g., more than 10% in QPE) compared to the base method. The similar results are mostly due to the fact that both methods do not consider operation-specific information. On the other hand, UREQA++ performs much better across different quantum algorithms, generally resulting in over 5-15% improvement in the error rate compared to the base method (e.g., $BV2$, QPE, HR_x^8H). UREQA++ is able to achieve this low circuit error by producing output state probabilities close to the correct output. Fig. 12 provides an example of this for QPE, where evidently, the circuits produced by the base method and UREQA have more error from the correct output (Table 5) than UREQA++. This demonstrates that the improved prediction quality with UREQA++ when using operation-specific information results in the selection of better circuit maps, which ultimately reduces the side-effect of erroneous quantum operations on current real quantum hardware.

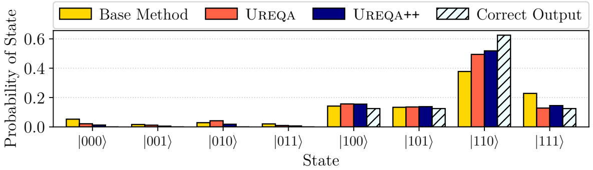


Figure 12: Overall output is much less erroneous with UREQA++ and UREQA than with the base method.

Table 5: State error results for the QPE output in Fig. 12.

Method	Error of Each State from the Correct Output (%)								Overall
	$\|000\rangle$	$\|001\rangle$	$\|010\rangle$	$\|011\rangle$	$\|100\rangle$	$\|101\rangle$	$\|110\rangle$	$\|111\rangle$	Error (%)
Base	5.3	1.6	2.9	2.0	1.7	0.9	24.7	10.3	$\sum/2=24.7$
UREQA	2.1	1.2	4.2	1.0	3.2	1.1	13.1	0.3	$\sum/2=13.1$
UREQA++	1.3	0.5	1.8	0.7	3.0	1.3	10.7	2.0	$\sum/2=10.7$

4 Related Work and Conclusion

Quantum Error Correction. Previous works have proposed algorithms for error correction in qubits which rely on heavy computational requirements which are unavailable in the current NISQ computers [4, 7, 11, 14, 24, 29, 30]. Thus, these methods are not applicable to NISQ technology, while UREQA++ helps reduce the error rate of quantum algorithms executed on NISQ hardware.

Minimization of Error Rates. Many recent works have employed approaches to mitigate the effects of error rates in quantum circuits via both online and offline methods [3, 9, 10, 12, 15, 17–19, 22, 23, 23, 26, 27, 31, 33, 34]. These include optimizing circuit maps to minimize error rates [23], migrating circuits to less-erroneous qubits [34], and minimizing error-prone operations [31]. UREQA's operation-aware circuit mapping technique achieves up to 15% reduction in error rate for quantum programs, compared to the current approaches. UREQA's open-source contribution pushes the state-of-the-art in quantum error rate prediction to minimize erroneous output, which can be leveraged by the quantum computing systems community.

Acknowledgment We are thankful to anonymous reviewers and our shepherd, Srinivasan Narayanamurthy, for improving the quality of this paper.

References

[1] Aha, D. W., Kibler, D., and Albert, M. K. Instance-based Learning Algorithms. *Machine learning 6*, 1 (1991), 37–66.

[2] Aleksandrowicz, et al. Qiskit: An Open-source Framework for Quantum Computing.(2019).

[3] Ash-Saki, et al. QURE: Qubit Re-allocation in Noisy Intermediate-Scale Quantum Computers. DAC.

[4] Bennett, et al. Mixed-State Entanglement and Quantum Error Correction. *Physical Review A 54*, 5 (1996).

[5] Bernstein, E., and Vazirani, U. Quantum Complexity Theory. *SIAM Journal on computing 26*, 5 (1997).

[6] Bravyi, et al. Trading Classical and Quantum Computational Resources. *Physical Review X 6*, 2 (2016).

[7] Burnett, et al. Decoherence Benchmarking of Superconducting Qubits. *npj Quantum Information 5*, 1 (2019).

[8] Cleve, R., Ekert, A., Macchiavello, C., and Mosca, M. Quantum Algorithms Revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences 454*, 1969 (Jan 1998).

[9] Das, et al. A Case for Multi-Prog. Quantum Comps. MICRO.

[10] Gokhale, et al. Partial Compilation of Variational Algorithms for Noisy Intermediate-Scale Quantum Machines. MICRO.

[11] Huang, et al. Performance of Quantum Error Correction with Coherent Errors. *Physical Review A 99*, 2 (2019).

[12] Huang, et al. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. ISCA.

[13] Koiran, P., Nesme, V., and Portier, N. A Quantum Lower Bound for the Query Complexity of Simon's Problem. In *International Colloquium on Automata, Languages, and Programming* (2005), Springer.

[14] Layden, et al. Ancilla-Free Quantum Error Correction Codes for Quantum Metrology. *Physical review letters 122*, 4 (2019).

[15] Li, G., Ding, Y., and Xie, Y. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. ASPLOS.

[16] Martonosi, et al. Next Steps in Quantum Computing: Computer Science's Role. *arXiv preprint arXiv:1903.10541* (2019).

[17] Mavadia, et al. Prediction and Real-Time Compensation of Qubit Decoherence via Machine Learning. *Nature communications 8* (2017).

[18] Murali, et al. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. ASPLOS.

[19] Murphy, et al. Controlling Error Orientation to Improve Quantum Algorithm Success Rates. *Physical Review A 99*, 3 (2019), 032318.

[20] Preskill, J. Quantum Computing in the NISQ Era and Beyond. *Quantum 2* (2018).

[21] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and De Freitas, N. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE 104*, 1 (2015), 148–175.

[22] Shi, et al. Optimized Compilation of Aggregated Instructions for Realistic Quantum Computers. ASPLOS.

[23] Smith, K. N., and Thornton, M. A. A Quantum Computational Compiler and Design Tool for Technology-Specific Targets. ISCA.

[24] Sun, et al. Experimental Quantum Error Correction with Binomial Bosonic Codes. In *APS Meeting Abstracts* (2019).

[25] Tannu, et al. Ensemble of Diverse Mappings: Improving Reliability of Quantum Comps. by Orchestrating Dissimilar Mistakes. MICRO.

[26] Tannu, et al. Mitigating Measurement Errors in Quantum Computers by Exploiting State-Dependent Bias. MICRO.

[27] Tannu, et al. Not All Qubits are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. ASPLOS.

[28] Tannu, S. S., Myers, Z. A., Nair, P. J., Carmean, D. M., and Qureshi, M. K. Taming the instruction bandwidth of quantum computers via hardware-managed error correction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), IEEE, pp. 679–691.

[29] Terhal, et al. Scalable Quantum Error Correction with the Bosonic GKP Code. In *APS Meeting Abstracts* (2019).

[30] Vuillot, et al. Quantum Error Correction with the Toric Gottesman-Kitaev-Preskill Code. *Physical Review A 99*, 3 (2019).

[31] Wille, et al. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. DAC.

[32] Zhang, P. Model Selection via Multifold Cross Validation. *The Annals of Statistics* (1993), 299–313.

[33] Zulehner, et al. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE TCAD* (2018).

[34] Zulehner, A., and Wille, R. Compiling SU (4) Quantum Circuits to IBM QX Architectures. ASPDAC.

Austere Flash Caching with Deduplication and Compression

Qiuping Wang[†], Jinhong Li[†], Wen Xia[‡], Erik Kruus[*], Biplob Debnath[*], and Patrick P. C. Lee[†]
[†]*The Chinese University of Hong Kong* [‡]*Harbin Institute of Technology, Shenzhen* [*]*NEC Labs*

Abstract

Modern storage systems leverage flash caching to boost I/O performance, and enhancing the space efficiency and endurance of flash caching remains a critical yet challenging issue in the face of ever-growing data-intensive workloads. Deduplication and compression are promising data reduction techniques for storage and I/O savings via the removal of duplicate content, yet they also incur substantial memory overhead for index management. We propose AustereCache, a new flash caching design that aims for memory-efficient indexing, while preserving the data reduction benefits of deduplication and compression. AustereCache emphasizes austere cache management and proposes different core techniques for efficient data organization and cache replacement, so as to eliminate as much indexing metadata as possible and make lightweight in-memory index structures viable. Trace-driven experiments show that our AustereCache prototype saves 69.9-97.0% of memory usage compared to the state-of-the-art flash caching design that supports deduplication and compression, while maintaining comparable read hit ratios and write reduction ratios and achieving high I/O throughput.

1 Introduction

High I/O performance is a critical requirement for modern data-intensive computing. Many studies (e.g., [1, 6, 9, 11, 20, 21, 24, 26, 31, 34, 35, 37]) propose solid-state drives (SSDs) as a flash caching layer atop hard-disk drives (HDDs) to boost performance in a variety of storage architectures, such as local file systems [1], web caches [20], data centers [9], and virtualized storage [6]. SSDs offer several attractive features over HDDs, including high I/O throughput (in both sequential and random workloads), low power consumption, and high reliability. In addition, SSDs have been known to incur much less cost-per-GiB than main memory (DRAM) [27], and such a significant cost difference still holds today (see Table 1). On the other hand, SSDs pose unique challenges over HDDs, as they not only have smaller available capacity, but also have poor endurance due to wear-out issues. Thus, in order to support high-performance workloads, caching as many objects as possible, while mitigating writes to SSDs to avoid wear-outs, is a paramount concern.

We explore both deduplication and compression as data reduction techniques for removing duplicate content on the I/O path, so as to mitigate both storage and I/O costs. Deduplication and compression target different granularities of data reduction and are complementary to each other: while deduplication removes chunk-level duplicates in a coarse-grained but lightweight manner, compression removes byte-level duplicates within chunks for further storage savings. With the ever-increasing growth of data in the wild, deduplication and/or compression have been widely adopted in primary [18, 23, 36] and backup [40, 42] storage systems. In particular, recent studies [24, 26, 37] augment flash caching with deduplication and compression, with emphasis on managing variable-size cached data in large replacement units [24] or designing new cache replacement algorithms [26, 37].

Type	Brand	Cost-Per-GiB ($)
DRAM	Crucial DDR4-2400 (16 GiB)	3.75
SSD	Intel SSD 545s (512 GiB)	0.24
HDD	Seagate BarraCuda (2 TiB)	0.025

Table 1: Cost-per-GiB of DRAM, SSD, and HDD based on the price quotes in January 2020.

Despite the data reduction benefits, existing approaches [24,26,37] of applying deduplication and compression to flash caching inevitably incur substantial memory overhead due to expensive index management. Specifically, in conventional flash caching, we mainly track the logical-to-physical address mappings for the flash cache. With both deduplication and compression enabled, we need dedicated index structures to track: (i) the mappings of each logical address to the physical address of the non-duplicate chunk in the flash cache after deduplication and compression, (ii) the cryptographic hashes (a.k.a. fingerprints (§2.1)) of all stored chunks in the flash cache for duplicate checking in deduplication, and (iii) the lengths of all compressed chunks that are of variable size. It is desirable to keep all such indexing metadata in memory for high performance, yet doing so aggravates the memory overhead compared to conventional flash caching. The additional memory overhead, which we refer to as *memory amplification*, can reach at least 16× (§2.3) and unfortunately compromise the data reduction effectiveness of deduplication and compression in flash caching.

In this paper, we propose AustereCache, a memory-efficient flash caching design that employs deduplication and compression for storage and I/O savings, while substantially mitigating the memory overhead of index structures in similar designs. AustereCache advocates *austere* cache management on the data layout and cache replacement policies to limit the memory amplification due to deduplication and compression. It builds on three core techniques: (i) *bucketization*, which achieves lightweight address mappings by deterministically mapping chunks into fixed-size buckets; (ii) *fixed-size compressed data management*, which avoids tracking chunk lengths in memory by organizing variable-size compressed chunks as fixed-size subchunks; and (iii) *bucket-based cache replacement*, which performs memory-efficient cache replacement on a per-bucket basis and leverages a compact sketch data structure [13] to track deduplication and recency patterns in limited memory space for cache replacement decisions.

We implement an AustereCache prototype and evaluate it through testbed experiments using both real-world and synthetic traces. Compared to CacheDedup [26], a state-of-the-art flash caching system that also supports deduplication and compression, AustereCache uses 69.9-97.0% less memory than CacheDedup, while maintaining comparable read hit ratios and write reduction ratios (i.e., it maintains the I/O performance gains through flash caching backed by deduplication and compression). In addition, AustereCache incurs limited CPU overhead on the I/O path, and can further boost I/O throughput via multi-threading.

The source code of our AustereCache prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/austerecache**.

2 Background

We first provide deduplication and compression background (§2.1). We then present a general flash caching architecture that supports deduplication and compression (§2.2), and show how such an architecture incurs huge memory amplification (§2.3). We finally argue that state-of-the-art designs are limited in mitigating the memory amplification issue (§2.4).

2.1 Deduplication and Compression

Deduplication and compression are data reduction techniques that remove duplicate content at different granularities.

Deduplication. We focus on *chunk-based deduplication*, which divides data into non-overlapping data units called *chunks* (of size KiB). Each chunk is uniquely identified by a *fingerprint (FP)* computed by some cryptographic hash (e.g., SHA-1) of the chunk content. If the FPs of two chunks are identical (or distinct), we treat both chunks as duplicate (or unique) chunks, since the probability that two distinct chunks have the same FP is practically negligible. Deduplication stores only one copy of duplicate chunks (in physical space), while referring all duplicate chunks (in logical space) to the copy via small-size pointers. Also, it keeps all mappings of FPs to physical chunk locations in an index structure used for duplicate checking and chunk lookups.

Chunk sizes may be fixed or variable. While content-based variable-size chunking generally achieves high deduplication savings due to its robustness against content shifts [42], it also incurs high computational overhead. On the other hand, fixed-size chunks fit better into flash units and fixed-size chunking often achieves satisfactory deduplication savings [26]. Thus, this work focuses on fixed-size chunking.

Compression. Unlike deduplication, which provides coarse-grained data reduction at the chunk level, *compression* aims for fine-grained data reduction at the byte level by transforming data into more compact form. Compression is often applied to the unique chunks after deduplication, and the output compressed chunks are of variable-size in general. For high performance, we apply sequential compression (e.g., Ziv-Lempel algorithm [43]) that operates on the bytes of each chunk in a single pass.

2.2 Flash Caching

We focus on building an SSD-based flash cache to boost the I/O performance of HDD-based primary storage, by storing the frequently accessed data in the flash cache. Flash caching has been extensively studied and adopted in different storage architectures (§7). Existing flash caching designs, which we collectively refer to as *conventional flash caching*, mostly support both *write-through* and *write-back* policies for read-intensive and write-intensive workloads, respectively [22]; the write-back policy is viable for flash caching due to the persistent nature of SSDs. For write-through, each write is persisted to both the SSD and the HDD before completion; for write-back, each write is completed right after it is persisted to the SSD. To support either policy, conventional flash caching needs an SSD-HDD translation layer that maps each *logical block address (LBA)* in an HDD to a *chunk address (CA)* in the flash cache.

In this work, we explore how to augment conventional flash caching with deduplication and compression to achieve storage and I/O savings, so as to address the limited capacity and wear-out issues in SSDs. Figure 1 shows the architecture of a general flash caching system that deploys deduplication and compression. We introduce two index structures: (i) *LBA-index*, which tracks how each LBA is mapped to the FP of a chunk (the mappings are many-to-one as multiple LBAs may refer to the same FP), and (ii) *FP-index*, which tracks how each FP is mapped to the CA and the length of a compressed chunk (the mappings are one-to-one). Thus, each cache lookup triggers two index lookups: it finds the FP of an LBA via the LBA-index, and then uses the FP to find the CA and the length of a compressed chunk via the FP-index. We also maintain a *dirty list* to track the list of LBAs of recent writes in write-back mode.

We now elaborate the I/O workflows of the flash caching system in Figure 1. For each write, the system partitions the written data into fixed-size chunks, followed by deduplication and compression: it first checks if each chunk is a duplicate; if not, it further compresses the chunk and writes the compressed chunk to the SSD (the compressed chunks can be packed into large-size units for better flash performance and endurance [24]). It updates the entries in both the LBA-index and the FP-index accordingly based on the FP of the chunk; in write-through mode, it also stores the fixed-size chunk in the HDD in uncompressed form. For each read, the sys-

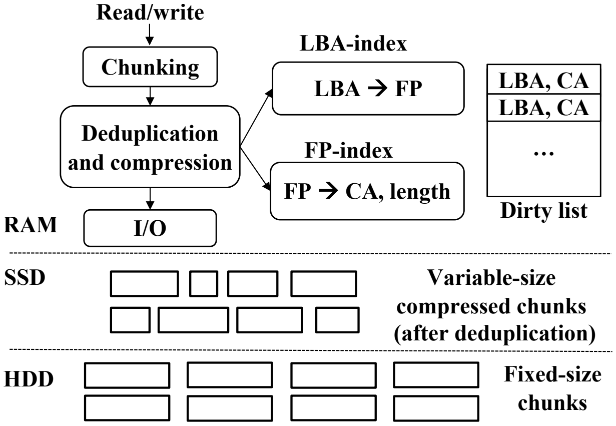


Figure 1: Architecture of a general flash caching system with deduplication and compression.

tem checks if the LBA is mapped to any existing CA via the lookups to both the LBA-index and the FP-index. If so (i.e., cache hit), the system decompresses and returns the chunk data; otherwise (i.e., cache miss), it fetches the chunk data from the HDD into the SSD, while it applies deduplication and compression to the chunk data as in a write.

2.3 Memory Amplification

While deduplication and compression intuitively reduce storage and I/O costs in flash caching by eliminating redundant content on the I/O path, both techniques inevitably incur significant memory costs for their index management. Specifically, if both index structures are entirely stored in memory for high performance, the memory usage is significant and much higher than that in conventional flash caching; we refer to such an issue as *memory amplification* (over conventional flash caching), which can negate the data reduction benefits of deduplication and compression.

We argue this issue through a simple analysis on the following configuration. Suppose that we deploy a 512 GiB SSD as a flash cache atop an HDD that has a working set of 4 TiB. Both the SSD and the HDD have 64-bit address space. For deduplication, we fix the chunk size as 32 KiB and use SHA-1 (20 bytes) for FPs. We also use 4 bytes to record the compressed chunk length. In the worst case, the LBA-index keeps 4 TiB / 32 KiB $= 128 \times 2^{20}$ (LBA, FP) pairs, accounting for a total of 3.5 GiB (each pair comprises an 8-byte LBA and a 20-byte FP). The FP-index keeps 512 GiB / 32 KiB $=$ 16×2^{20} (FP, CA) pairs, accounting for a total of 512 MiB (each pair comprises a 20-byte FP, an 8-byte CA, and a 4-byte length). The total memory usage of both the LBA-index and the FP-index is **4 GiB**. In contrast, conventional flash caching only needs to index 16×2^{20} (LBA, CA) pairs and the memory usage is **256 MiB**. This implies that flash caching with deduplication and compression amplifies the memory usage by **16**×. If we use a more collision-resistant hash function, the memory amplification is even higher; for example, it becomes **22.75**× if each FP is formed by SHA-256 (32 bytes). Note that our analysis does not consider other metadata for deduplication and compression (e.g., reference counts for deduplication), which further aggravates memory amplification over conventional flash caching.

In addition to memory amplification, deduplication and compression also add *CPU overhead* to the I/O path. Such overhead comes from: (i) the FP computation of each chunk, (ii) the compression of each chunk, and (iii) the lookups to both the LBA-index and the FP-index.

2.4 State-of-the-Art Flash Caches

We review two state-of-the-art flash caching designs, Nitro [24] and CacheDedup [26], both of which support deduplication and compression. We argue that both designs are still susceptible to memory amplification.

Nitro [24]. Nitro is the first flash cache that deploys deduplication and compression. To manage variable-size compressed chunks (a.k.a. extents [24]), Nitro packs them in large data units called *Write-Evict Units (WEUs)*, which serve as the basic units for cache replacement. The WEU size is set to align with the flash erasure block size for efficient garbage collection. When the cache is full, Nitro evicts a WEU based on the least-recently-used (LRU) policy. It manages index structures in DRAM (or NVRAM for persistence) to track all chunks in WEUs. If the memory capacity is limited, Nitro stores a *partial* FP-index in memory, at the expense that deduplication may miss detecting and removing some duplicates.

In addition to the memory amplification issue, organizing the chunks by WEUs may cause a WEU to include *stale chunks*, which are not referenced by any LBA in the LBA-index as their original LBAs may have been updated. Such stale chunks cannot be recycled immediately if their hosted WEUs also contain other valid chunks that are recently accessed due to the LRU policy, but instead occupy the cache space and degrade the cache hit ratio.

CacheDedup [26]. CacheDedup focuses on cache replacement algorithms that reduce the number of *orphaned entries*, which refer to either the LBAs that are in the LBA-index but have no corresponding FPs in the FP-index, or the FPs that are in the FP-index but are not referenced by any LBA. It proposes two deduplication-aware cache replacement policies, namely D-LRU and D-ARC, which augment the LRU and adaptive cache replacement (ARC) [29] policies, respectively. It also proposes a compression-enabled variant of D-ARC, called CD-ARC, which manages variable-size compressed chunks in WEUs as in Nitro [24]; note that CD-ARC suffers from the same stale-chunk issue as described above. CacheDedup maintains the same index structures as shown in Figure 1 (§2.2), in which the LBA-index stores LBAs to FPs, and the FP-index stores FPs to CAs and compressed chunk lengths. If it keeps both the LBA-index and the FP-index in memory for performance concerns, it still suffers from the same memory amplification issue. A follow-up work CDAC [37] improves the cache replacement of CacheDedup by incorporating ref-

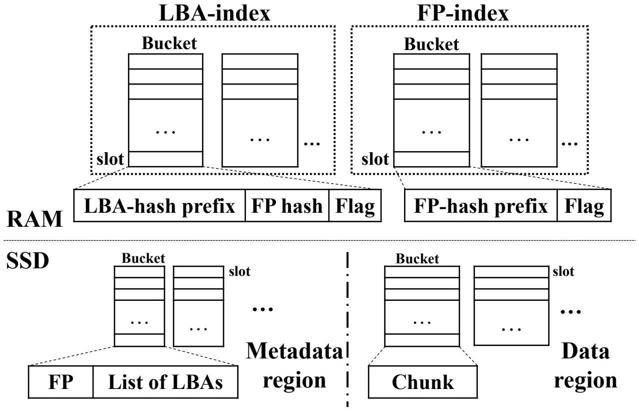


Figure 2: Bucketized data layouts of AustereCache in the LBA-index, the FP-index, as well as the metadata and data regions in flash.

erence counts and access patterns, but incurs even higher memory overhead for maintaining additional information.

3 AustereCache Design

AustereCache is a new flash caching design that leverages deduplication and compression to achieve storage and I/O savings as in prior work [24,26,37], but puts specific emphasis on reducing the memory usage for indexing. It aims for austere cache management via three key techniques.

- **Bucketization (§3.1).** To eliminate the overhead of maintaining address mappings in both the LBA-index and the FP-index, we leverage deterministic hashing to associate chunks with storage locations. Specifically, we hash index entries into equal-size partitions (called *buckets*), each of which keeps the *partial* LBAs and FPs for memory savings. Based on the bucket locations, we further map chunks into the cache space.
- **Fixed-size compressed data management (§3.2).** To avoid tracking chunk lengths in the FP-index, we treat variable-size compressed chunks as fixed-size units. Specifically, we divide variable-size compressed chunks into smaller fixed-size *subchunks* and manage the subchunks without recording the compressed chunk lengths.
- **Bucket-based cache replacement (§3.3).** To increase the likelihood of cache hits, we propose cache replacement on a per-bucket basis. In particular, we incorporate recency and deduplication awareness based on *reference counts* (i.e., the counts of duplicate copies referencing each unique chunk) for effective cache replacement. However, tracking reference counts incurs non-negligible memory overhead. Thus, we leverage a fixed-size compact sketch data structure [13] for reference count estimation in limited memory space with bounded errors.

3.1 Bucketization

Figure 2 shows the bucketized data layouts of AustereCache in both index structures and the flash cache space. We now do not consider compression, which we address in §3.2.

AustereCache partitions both the LBA-index and the FP-index into equal-size *buckets* composed of a fixed number of equal-size *slots*. Each slot corresponds to an LBA and an FP in the LBA-index and the FP-index, respectively. In addition, AustereCache divides the flash cache space into a *metadata region* and a *data region* that store metadata information and cached chunks, respectively; each region is again partitioned into buckets with multiple slots. Note that both regions are allocated the same numbers of buckets and slots as in the FP-index, such that each slot in the FP-index is a one-to-one mapping to the same slots in the metadata and data regions.

To reduce memory usage, each slot stores only the *prefix* of a key, rather than the full key. AustereCache first computes the hashes of both the LBA and the FP, namely *LBA-hash* and *FP-hash*, respectively. It stores the prefix bits of the LBA-hash and the FP-hash as the primary keys in one of the slots of a bucket in the LBA-index and the FP-index, respectively. Keeping only partial keys leads to hash collisions for different LBAs and FPs. To resolve hash collisions, AustereCache maintains the full LBA and FP information in the metadata region in flash, and any hash collision only leads to a cache miss without data loss. Also, by choosing proper prefix sizes, the collision rate should be low. AustereCache currently fixes 128 slots per bucket, mainly for efficient cache replacement (§3.3). For 16-bit prefixes as primary keys, the hash collision rate is only $1-(1-\frac{1}{2^{16}})^{128} \approx 0.2\%$, which is sufficiently low.

Write path. To write a unique chunk identified by an $(\mathrm{LBA},\mathrm{FP})$ pair to the flash cache, AustereCache updates both the LBA-index and the FP-index as follows. For the LBA-index, it uses the suffix bits of the LBA-hash to identify the bucket (e.g., for 2^k buckets, we check the k-bit suffix). It scans all slots in the corresponding bucket to see if the LBA-hash prefix has already been stored; otherwise, it stores the entry in an empty slot or evicts the least-recently-accessed slot if the bucket is full (see cache replacement in §3.3). It writes the following to the slot: the LBA-hash prefix (primary key), the FP-hash, and a valid flag that indicates if the slot stores valid data. Similarly, for the FP-index, it identifies the bucket and the slot using the FP-hash, and writes the FP-hash prefix (primary key) and the valid flag to the corresponding slot.

Based on the bucket and slot locations in the FP-index, AustereCache identifies the corresponding buckets and slots in the metadata and data regions of the flash cache. For the metadata region, it stores the complete FP and the list of LBAs; note that the same FP may be shared by multiple LBAs due to deduplication. We now fix the slot size as 512 bytes. If the slot is full and cannot store more LBAs, we evict the oldest LBA using FIFO to accommodate the new one. For the data region, AustereCache stores the chunk in the corresponding slot, which is also the CA.

Deduplication path. To perform deduplication on a written chunk identified by an $(\mathrm{LBA},\mathrm{FP})$ pair, AustereCache first identifies the bucket of the FP-index using the suffix bits of the FP-hash, and then searches for any slot that matches the same FP-hash prefix. If a slot is found, AustereCache checks the corresponding slot in the metadata region in flash and verifies if the input FP matches the one in the slot. If so, it means that a duplicate chunk is found, so AustereCache appends the LBA to the LBA list if the LBA does not exist before; otherwise, it implies an FP-hash prefix collision. When such a collision occurs, AustereCache invalidates the collided FP in the metadata region in flash and writes the chunk as described above (recall that the collision is unlikely from our calculation).

Read path. To read a chunk identified by an LBA, AustereCache first queries the LBA-index for the FP-hash using the LBA-hash prefix, followed by querying the FP-index for the slot that contains the FP-hash prefix. It then checks the corresponding slot of the metadata region in flash if an LBA is found in the LBA list. If so, the read is a cache hit and AustereCache returns the chunk from the data region; otherwise, the read is a cache miss and AustereCache accesses the chunk in the HDD via the LBA.

Analysis. We show via a simple analysis that the bucketization design of AustereCache has low memory usage. Suppose that we use a 512 GiB SSD as the flash cache with a 4 TiB working set of an HDD. We fix the chunk size as 32 KiB. Since each bucket has 128 slots, the LBA-index needs at most 2^{20} buckets to reference all chunks in the HDD, while the FP-index needs at most 2^{17} buckets to reference all chunks in the SSD. In addition, we store the first 16 prefix bits of both the LBA-hash and the FP-hash as the partial keys in the LBA-index and the FP-index, respectively. Since we use suffix bits to identify a bucket, we need 20 and 17 suffix bits to identify a bucket in the LBA-index and the FP-index, respectively. Thus, we configure an LBA-hash with $16+20=36$ bits and an FP-hash with $16+17=33$ bits.

We now compute the memory usage of each index structure, to which we apply bit packing for memory efficiency. For the LBA-index, each slot consumes 50 bits (i.e., a 16-bit LBA-hash prefix, a 33-bit FP-hash, and a 1-bit valid flag), so the memory usage of the LBA-index is $2^{20} \times 128 \times 50$ (bits) $=$ 800 MiB. For the FP-index, each slot consumes 17 bits (i.e., a 16-bit FP-hash prefix and a 1-bit valid flag), so the memory usage of the FP-index is $2^{17} \times 128 \times 17$ (bits) $=$ 34 MiB. The total memory usage of both index structures is 834 MiB, which is only around 20% of the 4 GiB memory space in the baseline (§2.3). While we do not consider compression, we emphasize that even with compression enabled, the index structures incur no extra overhead (§3.2).

Comparisons with other data structures. We may construct the LBA-index and the FP-index using other data structures for further memory savings. As an example, we consider the B+-tree [12], which is a balanced tree structure that organizes all leaf nodes at the same level. Suppose that we store index mappings in the leaf nodes that reside in flash, while the non-leaf nodes are kept in memory for referencing the leaf nodes. We evaluate the memory usage of the LBA-index and the FP-index as follows.

Suppose that each leaf node is mapped to a 4 KiB SSD page. For the LBA-index, each leaf node stores at most $\lfloor \frac{4096}{8+20} \rfloor = 146$ (LBA, FP) pairs (for an 8-byte LBA and a 20-byte FP). Referencing each leaf node takes 16 bytes (including an 8-byte LBA key and an 8-byte pointer). As there are 128×2^{20} (LBA, FP) pairs, the memory usage of the LBA-index is $\frac{128\times 2^{20}}{146} \times 16 \approx 14.0$ MiB (note that we exclude the memory usage for referencing non-leaf nodes). For the FP-index, each leaf node stores at most $\frac{4096}{20+8+4} = 128$ (FP, CA) pairs (for a 20-byte FP, an 8-byte CA, and a 4-byte length). Referencing each leaf node takes 28 bytes (including a 20-byte FP key and an 8-byte pointer). As there are 16×2^{20} (FP, CA) pairs, the memory usage of the FP-index is 3.5 MiB. Both the LBA-index and the FP-index incur much less memory usage than our current bucketization design (see above).

We can further use an in-memory Bloom Filter [8] to query for the existence of index mappings. For an error rate of 0.1%, each mapping uses 14.4 bits in a Bloom Filter. To track both 128×2^{20} (LBA, FP) pairs in the LBA-index and 16×2^{20} (FP, CA) pairs in the FP-index, we need an additional memory usage of 259.2 MiB.

We can conduct similar analyses for other data structures. For example, for the LSM-tree [32], we can maintain an in-memory structure to reference the on-disk LSM-tree nodes (a.k.a. *SSTables* [33]) that store the index mappings for the LBA-index and the FP-index. Then we can accordingly compute the memory usage for the LBA-index and the FP-index.

Even though these data structures support memory-efficient indexing, they incur additional flash access overhead. First, using B+-trees or LSM-trees for both the LBA-index and the FP-index incurs two flash accesses (one for each index structure) for indexing each chunk, while AustereCache issues only one flash access in the metadata region. Also, both the B+-tree and the LSM-tree have high write amplification [33] that degrades I/O performance. For these reasons, and perhaps more importantly, the synergies with compressed data management and cache replacement (see the following subsections), we settle on our proposed bucketized index design.

3.2 Fixed-Size Compressed Data Management

AustereCache can compress each unique chunk after deduplication for further space savings. To avoid tracking the length of the compressed chunk (which is of variable-size) in the index structures, AustereCache slices a compressed chunk into fixed-size *subchunks*, while the last subchunk is padded to fill a subchunk size. For example, for a subchunk size of 8 KiB, we store a compressed chunk of size 15 KiB as two subchunks, with the last subchunk being padded.

AustereCache allocates the same number of consecutive slots as that of subchunks in the FP-index (and hence the metadata and data regions in flash) to organize all subchunks

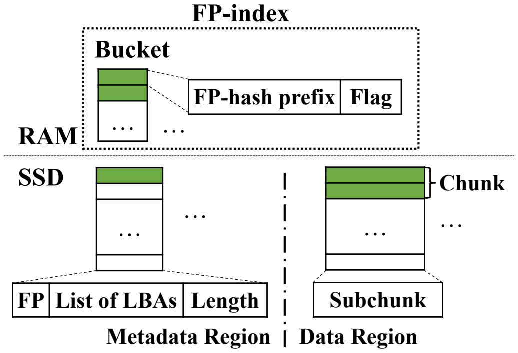


Figure 3: Fixed-size compressed data management, in which multiple consecutive slots are used for handling multiple fixed-size subchunks of a compressed chunk.

of a compressed chunk; note that the LBA-index remains unchanged, and each of its slots still references a chunk. Figure 3 shows an example in which a chunk is stored as two subchunks. For the FP-index, each of the two slots stores the corresponding FP-hash prefix, with an additional 1-bit valid flag indicating that the slot stores valid data. For the metadata region, it also allocates two slots, in which the first slot stores not only the full FP and the list of LBAs (§3.1), but also the length of the compressed chunk, while the second slot can be left empty to avoid redundant flash writes. For the data region, it allocates two slots for storing the two subchunks. Note that our design incurs no memory overhead for tracking the length of the compressed chunk in any index structure.

The read/write workflows with compression are similar to those without compression (§3.1), except that AustereCache now finds consecutive slots in the FP-index for the multiple subchunks of a compressed chunk. Note that we still keep 128 slots per bucket. However, since each slot now corresponds to a smaller-size subchunk, we need to allocate more buckets in the FP-index as well as the metadata and data regions in flash (the number of buckets in the LBA-index remains unchanged since each slot in the LBA-index still references a chunk). As we allocate more buckets for the FP-index, the memory usage also increases. Nevertheless, AustereCache still achieves memory savings for varying subchunk sizes (§5.4).

3.3 Bucket-Based Cache Replacement

Implementing cache replacement often requires priority-based data structures that decide which cached items should be kept or evicted, yet such data structures incur additional memory overhead. AustereCache opts to implement per-bucket cache replacement, i.e., the cache replacement decisions are based on only the entries within each bucket. It then implements specific cache replacement policies that incur no or limited additional memory overhead. Since each bucket is now configured with 128 slots, making the cache replacement decisions also incurs limited performance overhead.

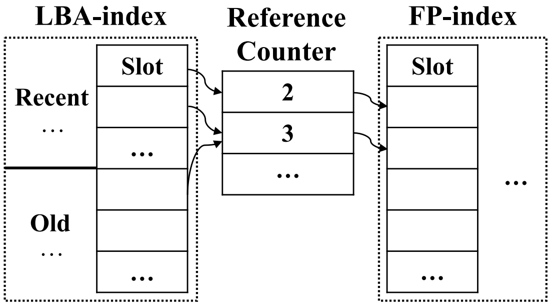


Figure 4: Cache replacement in the FP-index. When a bucket in the FP-index is full, the slot with the least reference counts (e.g. the slot with reference count 2) will be evicted.

For the LBA-index, AustereCache implements a bucket-based least-recently-used (LRU) policy. Specifically, each bucket sorts all slots by the recency of their LBAs, such that the slots at the lower offsets correspond to the more recently accessed LBAs (and vice versa). When the slot of an existing LBA is accessed, AustereCache shifts all slots at lower offsets than the accessed slot by one, and moves the accessed slot to the lowest offset. When a new LBA is inserted, AustereCache stores the new LBA in the slot at the lowest offset and shifts all other slots by one; if the bucket is full, the slot at the highest offset (i.e., the least-recently-accessed slot) is evicted. Such a design does not incur any extra memory overhead for maintaining the recency information of all slots.

For the FP-index, as well as the metadata and data regions in flash, we incorporate both deduplication and recency awareness into cache replacement. First, to incorporate deduplication awareness, AustereCache tracks the reference count for each FP-hash (i.e., the number of LBAs that share the same FP-hash). For each LBA being added to (resp. deleted from) the LBA-index, AustereCache increments (resp. decrements) the reference count of the corresponding FP-hash. When inserting a new FP to a full bucket, it evicts the slot that has the lowest reference count among all the slots in the same bucket. It also invalidates the corresponding slots in both the metadata and data regions in flash.

Simple reference counting does not address recency. To also incorporate recency awareness, AustereCache divides each LBA bucket into *recent slots* at lower offsets and *old slots* at higher offsets (now being divided evenly by half), as shown in Figure 4. Each LBA in the recent (resp. old) slots contributes to a count of two (resp. one) to the reference counting. Specifically, each newly inserted LBA is stored in the recent slot at the lowest offset in the LBA-index (see above), so AustereCache increments the reference count of the corresponding FP-hash by two. If an LBA is demoted from a recent slot to an old slot or is evicted from the LBA-index, AustereCache decrements the reference count of the corresponding FP-hash by one; similarly, if an LBA is promoted from an old slot to a recent slot, AustereCache increments the reference count of the corresponding FP-hash by one.

Maintaining reference counts for all FP-hashes, however, incurs non-negligible memory overhead. AustereCache addresses this issue by maintaining a Count-Min Sketch [13] to track the reference counts in a fixed-size compact data structure with bounded errors. A Count-Min Sketch is a two-dimensional counter array with r rows of w counters each (where r and w are configurable parameters). It maps each FP-hash (via an independent hash function) to one of the w counters in each of the r rows, and increments or decrements the mapped counters based on our reference counting mechanism. AustereCache can estimate the reference count of an FP-hash using the minimum value of all mapped counters of the FP-hash. Depending on the values of r and w, the error bounds can be theoretically proven [13].

Currently, our implementation fixes $r = 4$ and w equal to the total number of slots in the LBA-index. We justify via a simple analysis that sketch-based reference counting achieves significant memory savings. Referring to the analysis in §3.1, each FP-hash has 33 bits. If we track the reference counts of all FP-hashes, we need 2^{33} counters. On the other hand, if we use a Count-Min sketch, we set $r = 4$ and $w = 2^{27}$ (the total number of slots in the LBA-index), so there are $r \times w = 2^{29}$ counters, which consume only 1/16 of the memory usage of tracking all FP-hashes.

Our bucket-based cache replacement design works at the slot level. By using reference counting to make cache replacement decisions, AustereCache can promptly evict any stale chunk that is not referenced by an LBA, as opposed to the WEU design in Nitro and CD-ARC of CacheDedup (§2.4).

4 Implementation

We implement an AustereCache prototype as a user-space block device in C++ on Linux; the user-space implementation (as in Nitro [24]) allows us to readily deploy fast algorithms and multi-threading for performance speedups. Specifically, our AustereCache prototype issues reads and writes to the underlying storage devices via `pread` and `pwrite` system calls, respectively. It uses SHA-1 from the Intel ISA-L Crypto library [3] for chunk fingerprinting, LZ4 [4] for lossless stream-based compression, and XXHash [5] for fast hash computations in the index structures. We also integrate the cache replacement algorithms in CacheDedup [26] into our prototype for fair comparisons (§5). Our prototype now contains around 4.5 K LoC.

We leverage multi-threading to issue multiple read/write requests in parallel for high performance. Specifically, we implement bucket-level concurrency, such that each read/write request needs to acquire an exclusive lock to access a bucket in both the LBA-index and the FP-index, while multiple requests can access different buckets simultaneously.

5 Evaluation

We experiment AustereCache using both real-world and synthetic traces. We consider two variants of AustereCache: (i) AC-D, which performs deduplication only without compression, and (ii) AC-DC, which performs both deduplication and compression. We compare AustereCache with the three cache replacement algorithms of CacheDedup [26]: D-LRU, D-ARC, and CD-ARC (§2.4) (recall that CD-ARC combines D-ARC with the WEU-based compressed chunk management in Nitro [24]). For consistent naming, we refer to them as *CD-LRU-D*, *CD-ARC-D*, and *CD-ARC-DC*, respectively (i.e., the abbreviation of CacheDedup, the cache replacement algorithm, and the deduplication/compression feature). We summarize our evaluation findings as follows.

- Overall, AustereCache reduces memory usage by 69.9-97.0% compared to CacheDedup (Exp#1). It achieves the memory savings via different design techniques (Exp#2).
- AC-D achieves higher read hit ratios than CD-LRU-D and comparable read hit ratios as CD-ARC-D, while AC-DC achieves higher read hit ratios than CD-ARC-DC (Exp#3).
- AC-DC writes much less data to flash than CD-LRU-D and CD-ARC-D, while writing slightly more data than CD-ARC-DC due to padding (§3.2) (Exp#4).
- AustereCache maintains its substantial memory savings for different chunk sizes and subchunk sizes (Exp#5). We also study how it is affected by the sizes of both the LBA-index and the FP-index (Exp#6).
- AustereCache achieves high I/O throughput for different access patterns (Exp#7), while incurring small CPU overhead (Exp#8). Its throughput further improves via multi-threading (Exp#9).

Traces	Working Set (GiB)	Unique Data (GiB)	Write-to-Read Ratio
WebVM	2.71	69.37	3.24
Homes	19.19	240.00	10.81
Mail	59.01	983.78	5.09

Table 2: Basic statistics of FIU traces in 32 KiB chunks.

5.1 Traces

Our evaluation is driven by two traces.

FIU [23]. The FIU traces are collected from three different services with diverse properties, namely *WebVM*, *Homes*, and *Mail*, for the web, NFS, and mail services, respectively. Each trace describes the read/write requests on different chunks (of size 4 KiB or 512 bytes each), each of which is represented as an MD5 fingerprint of the chunk content.

To accommodate different chunk sizes, we take each trace of 4 KiB chunks and perform two-phase trace conversion as in [24]. In the first phase, we identify the initial state of the disk by traversing the whole trace and recording the LBAs of all chunk reads; any LBA that does not appear is assumed to have a dummy chunk fingerprint (e.g., all zeroes). In the second phase, we regenerate the trace of the corresponding chunk size based on the LBAs and compute the new chunk fingerprints. For example, we form a 32 KiB chunk by concatenating eight contiguous 4 KiB chunks and calculating a

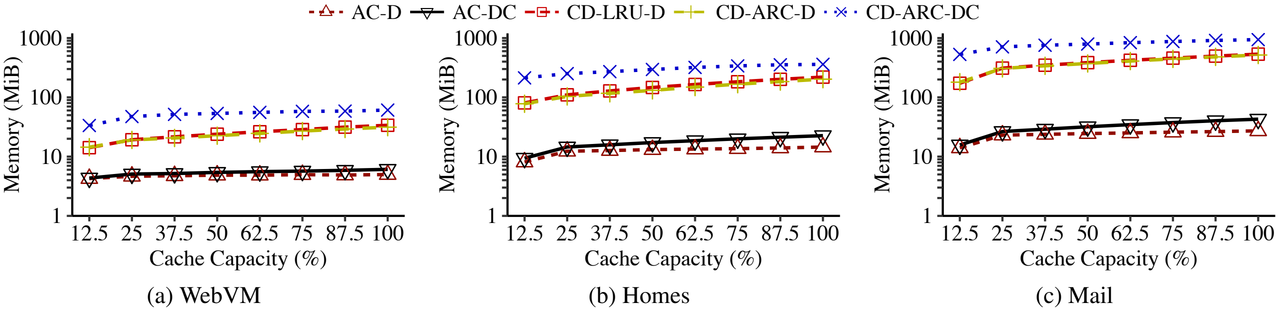


Figure 5: Exp#1 (Overall memory usage). Note that the y-axes are in log scale.

new SHA-1 fingerprint for the 32 KiB chunk. Table 2 shows the basic statistics of each regenerated FIU trace on 32 KiB chunks.

The original FIU traces have no compression details. Thus, for each chunk fingerprint, we set its *compressibility ratio* (i.e., the ratio of raw bytes to the compressed bytes) following a normal distribution with mean 2 and variance 0.25 as in [24].

Synthetic. For throughput measurement (§5.5), we build a synthetic trace generator to account for different access patterns. Each synthetic trace is configured by two parameters: (i) *I/O deduplication ratio*, which specifies the fraction of writes that can be removed on the write path due to deduplication; and (ii) *write-to-read ratio*, which specifies the ratios of writes to reads.

We generate a synthetic trace as follows. First, we randomly generate a working set by choosing arbitrary LBAs within the primary storage. Then we generate an access pattern based on the given write-to-read ratio, such that the write and read requests each follow a Zipf distribution. We derive the chunk content of each write request based on the given I/O deduplication ratio as well as the compressibility ratio as in the FIU trace generation (see above). Currently, our evaluation fixes the working set size as 128 MiB, the primary storage size as 5 GiB, and the Zipf constant as 1.0; such parameters are all configurable.

5.2 Setup

Testbed. We conduct our experiments on a machine running Ubuntu 18.04 LTS with Linux kernel 4.15. The machine is equipped with a 10-core 2.2 GHz Intel Xeon E5-2630v4 CPU, 32 GiB DDR4 RAM, a 1 TiB Seagate ST1000DM010-2EP1 SATA HDD as the primary storage, and a 128 GiB Intel SSDSC2BW12 SATA SSD as the flash cache.

Default setup. For both AustereCache and CacheDedup, we configure the size of the FP-index based on a fraction of the working set size (WSS) of each trace, and fix the size of the LBA-index four times that of the FP-index. We store both the LBA-index and the FP-index in memory for high performance. For AustereCache, we set the default chunk size and subchunk size as 32 KiB and 8 KiB, respectively. For CD-ARC-DC in CacheDedup, we set the WEU size as 2 MiB (the default in [26]).

5.3 Comparative Analysis

We compare AustereCache and CacheDedup in terms of memory usage, read hit ratios, and write reduction ratios using the FIU traces.

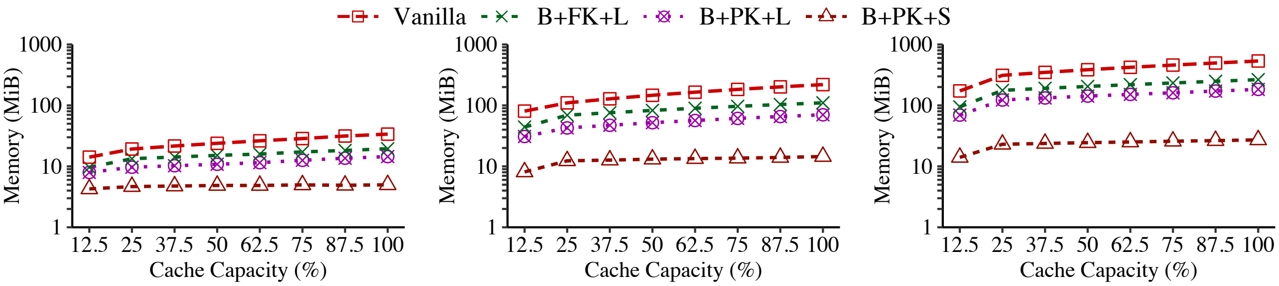
Exp#1 (Overall memory usage). We compare the memory usage of different schemes. We vary the flash cache size from 12.5% to 100% of WSS of each FIU trace, and configure the LBA-index and the FP-index based on our default setup (§5.2). To obtain the actual memory usage (rather than the allocated memory space for the index structures), we call `malloc_trim` at the end of each trace replay to return all unallocated memory from the process heap to the operating system, and check the residual set size (RSS) from `/proc/self/stat` as the memory usage.

Figure 5 shows that AustereCache significantly saves the memory usage compared to CacheDedup. For the non-compression schemes (i.e., AC-D, CD-LRU-D, and CD-ARC-D), AC-D incurs 69.9-94.9% and 70.4-94.7% less memory across all traces than CD-LRU-D and CD-ARC-D, respectively. For the compression schemes (i.e., AC-DC and CD-ARC-DC), AC-DC incurs 87.0-97.0% less memory than CD-ARC-DC.

AustereCache achieves higher memory savings than CacheDedup in compression mode, since CD-ARC-DC needs to additionally maintain the lengths of all compressed chunks, while AC-DC eliminates such information. If we compare the memory overhead with and without compression, CD-ARC-DC incurs 78-194% more memory usage than CD-ARC-D across all traces, implying that compression comes with high memory usage penalty in CacheDedup. On the other hand, AC-DC only incurs 2-58% more memory than AC-D.

Exp#2 (Impact of design techniques on memory savings). We study how different design techniques of AustereCache help memory savings. We mainly focus on bucketization (§3.1) and bucket-based cache replacement (§3.3); for fixed-size compressed data management (§3.2), we refer readers to Exp#1 for our analysis.

We choose CD-LRU-D of CacheDedup as our baseline and compare it with AC-D (both are non-compressed versions), and add individual techniques to see how they contribute to the memory savings of AC-D. We consider four variants:



(a) WebVM (b) Homes (c) Mail

Figure 6: Exp#2 (Impact of design techniques on memory savings).

- *Vanilla*. It refers to CD-LRU-D. It maintains the LRU lists that track the LBAs and FPs being accessed in the LBA index and the FP index, respectively.
- *B+FK+L*. It deploys bucketization (B), but keeps the full keys (FK) (i.e., LBAs and FPs) in each slot. Each bucket implements the LRU policy (L) independently and keeps an LRU list of the slot IDs being accessed.
- *B+PK+L*. It deploys bucketization (B) and now keeps the prefix keys (PK) in both the LBA-index and the FP-index. It still implements the LRU policy as in B+FK+L.
- *B+PK+S*. It deploys bucketization (B) and keeps the prefix keys (PK). It maintains reference counts in a sketch (S). Note that it is equivalent to AC-D.

Figure 6 presents the memory usage versus the cache capacity, where the memory usage is measured as in Exp#1. Compared to Vanilla, B+FK+L saves the memory usage by 30.6-50.6%, while B+PK+L further increases the savings to 43.9-68.0% due to keeping prefix keys in the index structures. B+FK+S (i.e., AC-D) increases the overall memory savings to 69.9-94.9% by keeping reference counts in a sketch as opposed to maintaining LRU lists with full LBAs and FPs.

Exp#3 (Read hit ratio). We evaluate different schemes with the *read hit ratio*, defined as the fraction of read requests that receive cache hits over the total number of read requests.

Figure 7 shows the results. AustereCache generally achieves higher read hit ratios than different CacheDedup algorithms. For the non-compression schemes, AC-D increases the read hit ratio of CD-LRU-D by up to 39.2%. The reason is that CD-LRU-D is only aware of the request recency and fails to clean stale chunks in time (§2.4), while AustereCache favors to evict chunks with small reference counts. On the other hand, AC-D achieves similar read hit ratios to CD-ARC-D, and in particular has a higher read hit ratio (up to 13.4%) when the cache size is small in WebVM (12.5% WSS) by keeping highly referenced chunks in cache. For the compression schemes, AC-DC has higher read hit ratios than CD-ARC-DC, by 0.5-30.7% in WebVM, 0.7-9.9% in Homes, and 0.3-6.2% in Mail. Note that CD-ARC-DC shows a lower read hit ratio than CD-ARC-D although it intuitively stores more chunks with compression, mainly because it cannot quickly evict stale chunks due to the WEU-based organization (§2.4).

Exp#4 (Write reduction ratio). We further evaluate different schemes in terms of the *write reduction ratio*, defined as the fraction of reduction of bytes written to the cache due to both deduplication and compression. A high write reduction ratio implies less written data to the flash cache and hence improved performance and endurance.

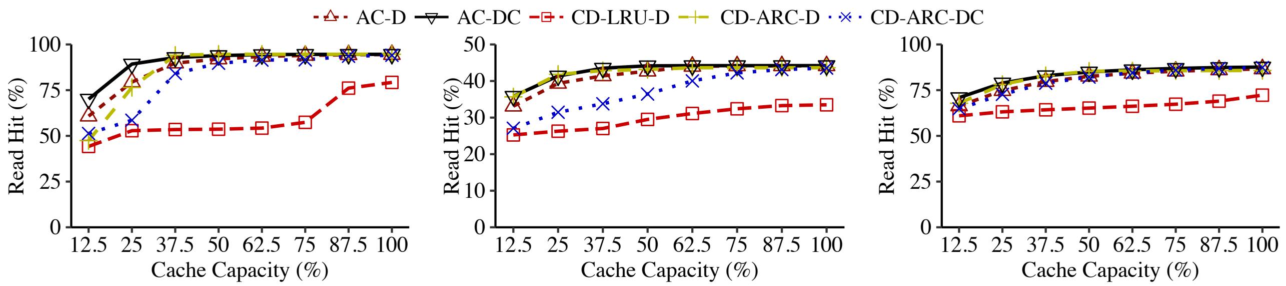
Figure 8 shows the results. For the non-compression schemes, AC-D, CD-LRU-D, and CD-ARC-D show marginal differences in WebVM and Homes, while in Mail, AC-D has lower write reduction ratios than CD-LRU-D by up to 17.5%. We find that CD-LRU-D tends to keep more stale chunks in cache, thereby saving the writes that hit the stale chunks. For example, when the cache size is 12.5% of WSS in Mail, 17.1% of the write reduction in CD-LRU-D comes from the writes to the stale chunks, while in WebVM and Homes, the corresponding numbers are only 3.6% and 1.1%, respectively. AC-D achieves lower write reduction ratios than CD-LRU-D, but achieves much higher read hit ratios by up to 39.2% by favoring to evict the chunks with small reference counts (Exp#3).

For the compression schemes, both CD-ARC-DC and AC-DC have much higher write reduction ratios than the non-compression schemes due to compression. However, AC-DC shows a slightly lower write reduction ratio than CD-ARC-DC by 7.7-14.5%. The reason is that AC-DC pads the last subchunk of each variable-size compressed chunk, thereby incurring extra writes. As we show later in Exp#5 (§5.4), a smaller subchunk size can reduce the padding overhead, although the memory usage also increases.

5.4 Sensitivity to Parameters

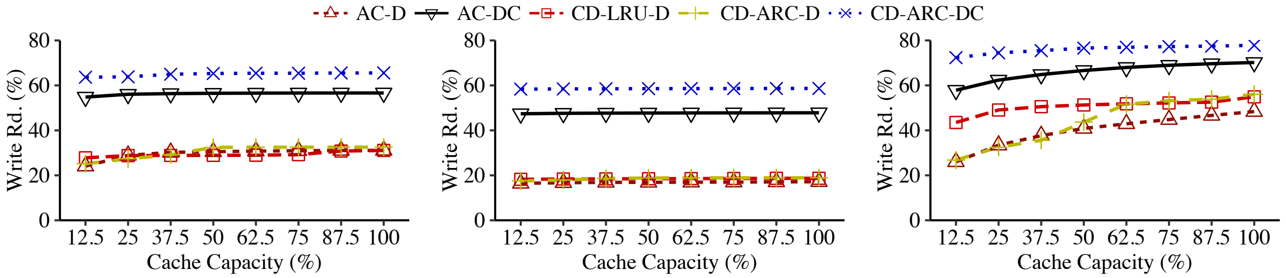
We evaluate AustereCache for different parameter settings using the FIU traces.

Exp#5 (Impact of chunk sizes and subchunk sizes). We evaluate AustereCache on different chunk sizes and subchunk sizes. We focus on the Homes trace and vary the chunk sizes and subchunk sizes as described in §5.1. For varying chunk sizes, we fix the subchunk size as one-fourth of the chunk size; for varying subchunk sizes, we fix the chunk size as 32 KiB. We focus on comparing AC-DC and CD-ARC-DC by fixing the cache size as 25% of WSS. Note that CD-ARC-DC is unaffected by the subchunk size.



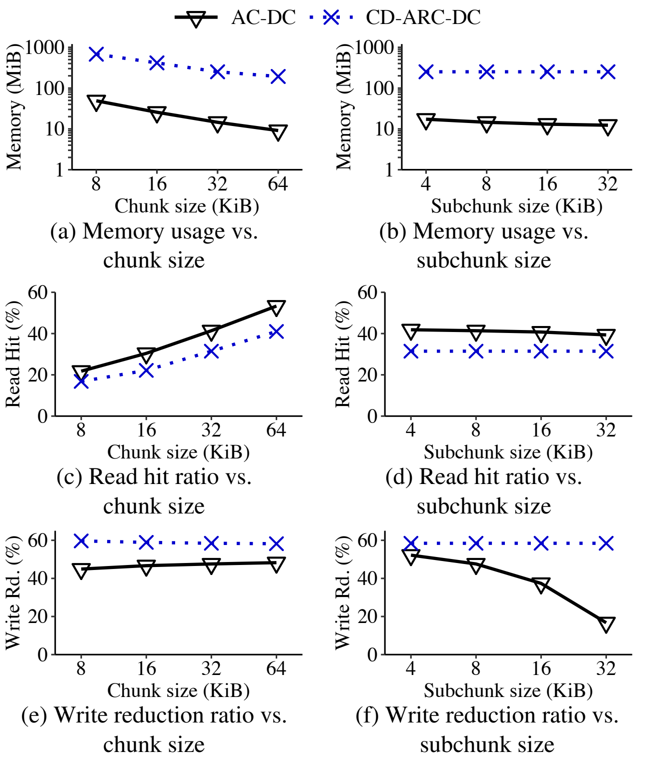
(a) WebVM (b) Homes (c) Mail

Figure 7: Exp#3 (Read hit ratio).



(a) WebVM (b) Homes (c) Mail

Figure 8: Exp#4 (Write reduction ratio).



(a) Memory usage vs. chunk size

(b) Memory usage vs. subchunk size

(c) Read hit ratio vs. chunk size

(d) Read hit ratio vs. subchunk size

(e) Write reduction ratio vs. chunk size

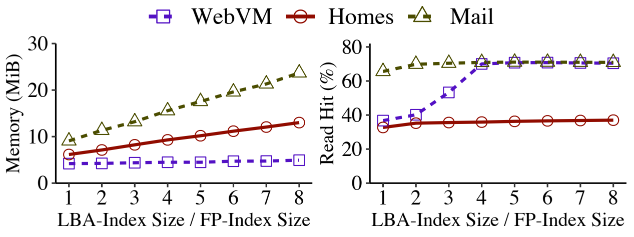
(f) Write reduction ratio vs. subchunk size

Figure 9: Exp#5 (Impact of chunk sizes and subchunk sizes). We focus on the Homes trace and fix the cache size as 25% of WSS in Homes.

AC-DC maintains the significant memory savings compared to CD-ARC-DC, by 92.8-95.3% for varying chunk sizes (Figure 9(a)) and 93.1-95.1% for varying subchunk sizes (Figure 9(b)). It also maintains higher read hit ratios than CD-ARC-DC, by 5.0-12.3% for varying chunk sizes (Figure 9(c)) and 7.9-10.4% for varying subchunk sizes (Figure 9(d)). AC-DC incurs a (slightly) less write reduction ratio than CD-ARC-DC due to padding, by 10.0-14.8% for varying chunk sizes (Figure 9(e)); the results are consistent with those in Exp#4. Nevertheless, using a smaller subchunk size can mitigate the padding overhead. As shown in Figure 9(f), the write reduction ratio of AC-DC approaches that of CD-ARC-DC when the subchunk size decreases. When the subchunk size is 4 KiB, AC-DC only has a 6.2% less write reduction ratio than CD-ARC-DC. Note that if we change the subchunk size from 8 KiB to 4 KiB, the memory usage increases from 14.5 MiB to 17.3 MiB (by 18.8%), since the number of buckets is doubled in the FP-index (while the LBA-index remains the same).

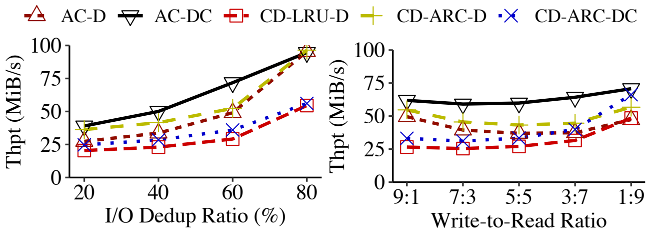
Exp#6 (Impact of LBA-index sizes). We study the impact of LBA-index sizes. We vary the LBA-index size from 1× to 8× of the FP-index size (recall that the default is 4×), and fix the cache size as 12.5% of WSS.

Figure 10 depicts the memory usage and read hit ratios; we omit the write reduction ratio as there is nearly no change for varying LBA-index sizes. When the LBA-index size increases, the memory usage increases by 17.6%, 111.5%, and 160.9% in WebVM, Homes and Mail, respectively (Figure 10(a)), as we allocate more buckets in the LBA-index. Note that the increase in memory usage in WebVM is less than those in Homes and Mail, mainly because the WSS of WebVM is small and incurs a small actual increase of the total memory usage. Also, the read hit ratio increases with



(a) Memory usage (b) Read hit ratio

Figure 10: Exp#6 (Impact of LBA-index sizes).



(a) Throughput vs. I/O dedup ratio (write-to-read ratio 7:3)

(b) Throughput vs. write-to-read ratio (I/O dedup ratio 50%)

Figure 11: Exp#7 (Throughput).

the LBA-index size, until the LBA-index reaches 4× of the FP-index size (Figure 10(b)). In particular, for WebVM, the read hit ratio grows from 36.7% (1×) to 70.4% (8×), while for Homes and Mail, the read hit ratios increase by only 4.3% and 5.3%, respectively. The reason is that when the LBA-index size increases, WebVM shows a higher increase in the total reference counts of the cached chunks than Homes and Mail, implying that more reads can be served by the cached chunks (i.e., higher read hit ratios).

5.5 Throughput and CPU Overhead

We measure the throughput and CPU overhead of AustereCache. We conduct the evaluation on synthetic traces for varying I/O deduplication ratios and write-to-read ratios. We focus on the write-back policy (§2.2), in which AustereCache first persists the written chunks to the flash cache and flushes the chunks to the HDD when they are evicted from the cache. We use direct I/O to remove the impact of page cache. We report the averaged results over five runs, while the standard deviations are small (less than 2.7%) and hence omitted.

Exp#7 (Throughput). We compare AustereCache and CacheDedup in throughput using synthetic traces. We fix the cache size as 50% of the 128 MiB WSS. Both systems work in single-threaded mode.

Figures 11(a) and 11(b) show the results for varying I/O deduplication ratios (with a fixed write-to-read ratio 7:3, which represents a write-intensive workload as in FIU traces) and varying write-to-read ratios (with a fixed I/O deduplication ratio 50%), respectively. For the non-compression schemes, AC-D achieves 18.5-86.6% higher throughput than CD-LRU-D for all cases except when the write-to-read ratio is 1:9 (slightly slower by 2.3%). Compared to CD-ARC-D, AC-D is slower by 1.1-24.5%, since both AC-D and CD-ARC-D have similar read hit ratios and write reduction ratios (§5.3), while AC-D issues additional reads and writes to the metadata region (CD-ARC-D keeps all indexing information in memory). AC-D achieves similar throughput to CD-ARC-D when there are more duplicate chunks (i.e., under high I/O deduplication ratios). For compression schemes, AC-DC achieves 6.8-99.6% higher throughput than CD-ARC-DC.

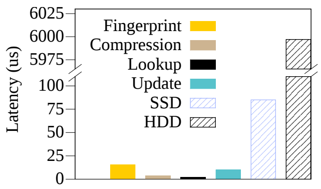


Figure 12: Exp#8 (CPU overhead).

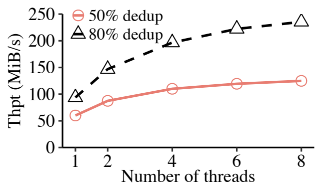


Figure 13: Exp#9 (Throughput of multi-threading).

Overall, AC-DC achieves the highest throughput among all schemes for two reasons. First, AustereCache generally achieves higher or similar read hit ratios compared to CacheDedup algorithms (§5.3). Second, AustereCache incorporates deduplication awareness into cache replacement by caching chunks with high reference counts, thereby absorbing more writes in the SSD and reducing writes to the slow HDD.

Exp#8 (CPU overhead). We study the CPU overhead of deduplication and compression of AustereCache along the I/O path. We measure the latencies of four computation steps, including fingerprint computation, compression, index lookup, and index update. Specifically, we run the WebVM trace with a cache size of 12.5% of WSS, and collect the statistics of 100 non-duplicate write requests. We also compare their latencies with those of 32 KiB chunk write requests to the SSD and the HDD using the `fio` benchmark tool [2].

Figure 12 depicts the results. Fingerprint computation has the highest latency (15.5 μs) among all four steps. In total, AustereCache adds around 31.2 μs of CPU overhead. On the other hand, the latencies of 32 KiB writes to the SSD and the HDD are 85 μs and 5,997 μs, respectively. Note that the CPU overhead can be suppressed via multi-threaded processing, as shown in Exp#9.

Exp#9 (Throughput of multi-threading). We evaluate the throughput gain of AustereCache when it enables multi-threading and issues concurrent requests to multiple buckets (§4). We use synthetic traces with a write-to-read ratio of 7:3, and consider the I/O deduplication ratio of 50% and 80%.

Figure 13 shows the throughput versus the number of threads being configured in AustereCache. When the number of threads increases, AustereCache shows a higher throughput gain under 80% I/O deduplication ratio (from 93.8 MiB/s to 235.5 MiB/s, or 2.51×) than under 50% I/O deduplication ratio (from 60.0 MiB/s to 124.9 MiB/s, or 2.08×). A higher I/O deduplication ratio implies less I/O to flash, and AustereCache benefits more from multi-threading on parallelizing the computation steps in the I/O path and hence sees a higher throughput gain.

6 Discussion

We discuss the following open issues of AustereCache.

Choices of chunk/subchunk sizes. AustereCache by default uses 32 KiB chunks and 8 KiB subchunks to align with common flash page sizes (e.g., 4 KiB or 8 KiB) in commodity SSDs, while preserving memory savings even for various chunk/subchunk sizes (Exp#5 in §5.4). Larger chunk/subchunk sizes reduce the chunk management overhead, at the expense of issuing more read-modify-write operations for small requests from upper-layer applications. Efficiently managing small chunks/subchunks in large-size I/O units in flash caching [24, 25], while maintaining memory efficiency in indexing, is future work.

Impact of indexing on flash endurance. AustereCache currently reduces its memory usage by keeping only limited indexing information in memory and full indexing details in flash (i.e., the metadata region). Since the indexing information generally has a smaller size than the cached chunks, we expect that the updates of the metadata region bring limited degradations to flash endurance, compared to the writes of chunks to the data region. An in-depth analysis of how AustereCache affects flash endurance is future work.

AustereCache assumes that the flash translation layer supports efficient flash erasure management (e.g., applying write combining before writing chunks to flash). To further mitigate the flash erasure overhead, one possible design extension is to adopt a log-structured data organization in flash in order to limit random writes, which are known to degrade flash endurance [30].

7 Related Work

Flash caching. Flash caching has been extensively studied to improve I/O performance. For example, Bcache [1] is a block-level cache for Linux file systems; FlashCache [20] is a file cache for web servers; Mercury [9] is a hypervisor cache for shared storage in data centers; CloudCache [6] estimates the demands of virtual machines (VMs) and manages cache space for VMs in virtualized storage.

Several studies focus on better flash caching management. For example, FlashTier [34] exploits caching workloads in cache block management; Kim *et al.* [21] exploit application hints to cache write requests; DIDACache [35] takes a software-hardware co-design approach to eliminate duplicate garbage collection. To improve the endurance of flash caching, Cheng *et al.* [11] propose erasure-aware heuristics to admit cache insertions; S-RAC [31] selectively evicts cache items based on temporal locality; Pannier [25] manages the flash cache in large-size units (called containers) with erasure awareness; Wang *et al.* [38] use machine learning to remove unnecessary writes to flash.

Deduplication and compression. AustereCache exploits deduplication and compression in flash caching. Extensive work has shown the effectiveness of deduplication and/or compression in storage and I/O savings in primary [18, 23, 36], backup [16, 40, 42], and memory storage [19, 39]. For flash storage, CAFTL [10] implements deduplication in the flash translation layer to reduce flash writes; SmartDedup [41] organizes in-memory and on-disk fingerprints for resource-constrained devices; FlaZ [28] applies transparent and online I/O compression for efficient flash caching. Prior studies [24, 26, 37] also exploit deduplication and compression in flash caching, but incur high memory overhead in metadata management (§2.4). On the other hand, AustereCache aims for memory efficiency without compromising the storage and I/O savings achieved by deduplication and compression.

Memory-efficient designs. Prior studies propose memory-efficient data structures for flash storage. ChunkStash [15] uses fingerprint prefixes to index fingerprints on SSDs in backup deduplication. SkimpyStash [14] designs a hash-table-based index that stores chained linked lists on SSDs for deduplication systems. SILT [27] uses partial-key hashing for efficient indexing in key-value stores. TinyLFU [17] uses Counting Bloom Filters to estimate item frequencies in cache admission. Our bucketization design (§3.1) is similar to the Quotient Filter (also used in flash caching [7]) in prefix-key matching. AustereCache specifically targets flash caching with deduplication and compression, and incorporates several techniques for high memory efficiency.

8 Conclusion

AustereCache makes a case of integrating deduplication and compression into flash caching while significantly mitigating the memory overhead due to indexing. It builds on three techniques to aim for austere cache management: (i) bucketization removes address mappings from indexing; (ii) fixed-size compressed data management removes compressed chunk lengths from indexing; and (iii) bucket-based cache replacement tracks reference counts in a compact sketch structure to achieve high read hit ratios. Evaluation on both real-world and synthetic traces shows that AustereCache achieves significant memory savings, with high read hit ratios, high write reduction ratios, and high throughput.

Acknowledgments: We thank our shepherd, William Jannen, and the anonymous reviewers for their comments. This work was supported in part by RGC of Hong Kong (AoE/P-404/18), NSFC (61972441), and the Shenzhen Science and Technology Program (JCYJ20190806143405318). The corresponding author is Wen Xia.

References

[1] Bcache: A linux kernel block layer cache. `http://bcache.evilpiepirate.org/`.

[2] Fio - Flexible I/O Tester Synthetic Benchmark. http://git.kernel.dk/?p=fio.git.

[3] ISA-L_crypto. https://github.com/intel/isa-l_crypto.

[4] LZ4. https://en.wikipedia.org/wiki/LZ4_(compression_algorithm).

[5] XXHash. https://github.com/Cyan4973/xxHash.

[6] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proc. of USENIX FAST*, 2016.

[7] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proc. of VLDB Endowment*, 5(11):1627–1637, 2012.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 12(7):422–426, 1970.

[9] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proc. of IEEE MSST*, 2012.

[10] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proc. of USENIX FAST*, 2011.

[11] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li. Erasing belady's limitations: In search of flash cache offline optimality. In *Proc. of USENIX ATC*, 2016.

[12] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[14] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proc. of ACM SIGMOD*, 2011.

[15] B. K. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. of USENIX ATC*, 2010.

[16] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *Proc. of USENIX ATC*, 2019.

[17] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A highly efficient cache admission policy. *ACM Trans. on Storage*, 13(4):1–31, 2017.

[18] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplicationlarge scale study and system design. In *Proc. of USENIX ATC*, 2012.

[19] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. S. Lui. SmartMD: A high performance deduplication engine with mixed pages. In *Proc. of USENIX ATC*, 2017.

[20] T. Kgil and T. Mudge. FlashCache: a NAND flash memory file cache for low power web servers. In *Proc. of ACM CASES*, 2006.

[21] S. Kim, H. Kim, S.-H. Kim, J. Lee, and J. Jeong. Request-oriented durable write caching for application performance. In *Proc. of USENIX ATC*, 2015.

[22] R. Koller, , L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of USENIX FAST*, 2013.

[23] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. on Storage*, 6(3):13, 2010.

[24] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proc. of USENIX ATC*, 2014.

[25] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Trans. on Storage*, 13(3):1–34, 2017.

[26] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao. CacheDedup: In-line deduplication for flash caching. In *Proc. of USENIX FAST*, 2016.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. of ACM SOSP*, 2011.

[28] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve SSD-based I/O caches. In *Proc. of ACM EuroSys*, 2010.

[29] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of USENIX FAST*, 2003.

[30] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: random write considered harmful in solid state drives. In *Proc. of USENIX FAST*, 2012.

[31] Y. Ni, J. Jiang, D. Jiang, X. Ma, J. Xiong, and Y. Wang. S-RAC: SSD friendly caching for data center workloads. In *Proc. of ACM Systor*, 2016.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[33] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[34] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proc. of ACM EuroSys*, 2012.

[35] Z. Shen, F. Chen, Y. Jia, and Z. Shao. DIDACache: A deep integration of device and application for flash based key-value caching. In *Proc. of USENIX FAST*, 2017.

[36] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *Proc. of USENIX FAST*, 2012.

[37] Y. Tan, J. Xie, C. Xu, Z. Yan, H. Jiang, Y. Zhao, M. Fu, X. Chen, D. Liu, and W. Xia. CDAC: Content-driven deduplication-aware storage cache. In *Proc. of MSST*, 2019.

[38] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou. Efficient SSD caching by avoiding unnecessary writes using machine learning. In *Proc. of ACM ICPP*, 2018.

[39] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proc. of USENIX FAST*, 2018.

[40] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proc. of USENIX ATC*, 2011.

[41] Q. Yang, R. Jin, and M. Zhao. SmartDedup: Optimizing deduplication for resource-constrained devices. In *Proc. of USENIX ATC*, 2019.

[42] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.

[43] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337 – 343, May 1977.

DADI Block-Level Image Service for Agile and Elastic Application Deployment

Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu and Windsor Hsu
Alibaba Group

Abstract

Businesses increasingly need agile and elastic computing infrastructure to respond quickly to real world situations. By offering efficient process-based virtualization and a layered image system, containers are designed to enable agile and elastic application deployment. However, creating or updating large container clusters is still slow due to the image downloading and unpacking process. In this paper, we present DADI Image Service, a block-level image service for increased agility and elasticity in deploying applications. DADI replaces the waterfall model of starting containers (downloading image, unpacking image, starting container) with fine-grained on-demand transfer of remote images, realizing instant start of containers. DADI optionally relies on a peer-to-peer architecture in large clusters to balance network traffic among all the participating hosts. DADI efficiently supports various kinds of runtimes including cgroups, QEMU, etc., further realizing "build once, run anywhere". DADI has been deployed at scale in the production environment of Alibaba, serving one of the world's largest ecommerce platforms. Performance results show that DADI can cold start 10,000 containers on 1,000 hosts within 4 seconds.

1 Introduction

As business velocity continues to rise, businesses increasingly need to quickly deploy applications, handle unexpected surge, fix security flaws, and respond to various real world situations. By offering efficient process-based virtualization and a layered image system, containers are designed to enable agile and elastic application deployment. However, creating or updating large container clusters is still slow due to the image downloading and unpacking process. For example, Verma et al. in [37] report that the startup latency of containers is highly variable with a typical median of about 25s, and pulling layers (packages) accounts for about 80% of the total time.

Highly elastic container deployment has also become expected of modern cloud computing platforms. In serverless computing [23], high cold-start latency could violate responsiveness SLAs. Workarounds for the slow start are cumbersome and expensive, and include storing all images on all possible hosts. Therefore, minimizing cold-start latency is considered a critical system-level challenge for serverless computing [14, 23].

There has been recent work on reducing container startup time by accelerating the image downloading process with a peer-to-peer (P2P) approach [20, 22, 24, 30, 37]. We relied on a P2P download tool for several years to cope with the scalability problem of the Container Registry. However, the startup latency was still unsatisfactory. Another general approach to the problem is to read data on-demand from remote images [9, 16, 18, 19, 21, 33, 41]. Because container images are organized as overlaid layers of files and are presented to the container runtime as a file system directory, all of the previous work adhered to the file system interface, even though some of them actually used block stores as their backends.

Implementing a POSIX-complaint file system interface and exposing it via the OS kernel is relatively complex. Moreover, using file-based layers has several disadvantages. First, updating big files (or their attributes) is slow because the system has to copy whole files to the writable layer before performing the update operations. Second, creating hard links is similarly slow, because it also triggers the copy action as cross layer references are not supported by the image. Third, files may have a rich set of types, attributes, and extended attributes that are not consistently supported on all platforms. Moreover, even on one platform, support for capabilities such as hard links, sparse files, etc. tends to be inconsistent across file systems.

With the rapid growth of users running containers on public cloud and hybrid cloud, virtualized secure containers are becoming mainstream. Although it is possible to pass a file-based image from host to guest via 9p [1] or virtio-fs [10], there is usually a performance cost. There are also complications in handling heterogeneous containers such as Windows guest on Linux host, or vice versa. This means that some users may not be able to burst efficiently to public clouds, i.e. run their applications primarily on premise with an efficient container runtime, and scale them out under load to public clouds with a virtualized secure container runtime.

In this paper, we observe that the benefits of a layered image are not contingent on representing the layers as sets of file changes. More specifically, we can achieve the same effect with block-based layers where each layer still corresponds to a set of file changes but is physically the set of changes at the block level underneath a given file system. Such a design allows the image service to be file system and platform agnostic. The image service is solely responsible for managing and distributing physical images to the appropriate hosts. It is up to the individual host or container on the host to interpret

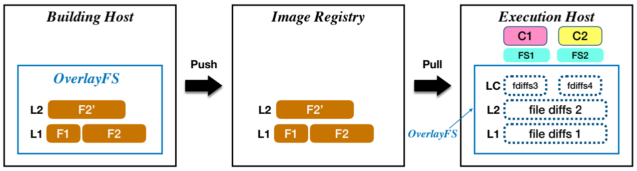


Figure 1: Layered Container Image. The image layers (L1, L2) are read-only shared by multiple containers (C1, C2) while the container layers (LC) are privately writable.

the image with an appropriate file system. This approach also allows dependency on the file system to be explicitly captured at image creation time, further enhancing consistency in the runtime environment of applications.

We have designed and implemented a complete system called *DADI Image Service* (*DADI* in short) based on this approach. The name *DADI* is an acronym for *Data Acceleration for Disaggregated Infrastructure* and describes several of our initiatives in enabling a disaggregated infrastructure. At the heart of the DADI Image Service is a new construct called Overlay Block Device (OverlayBD) which provides a merged view of a sequence of block-based layers. Conceptually, it can be seen as the counterpart of union file systems that are usually used to merge container images today. It is simpler than union file systems and this simplicity enables optimizations including flattening of the layers to avoid performance degradation for containers with many layers. More generally, the simplicity of block-based layers facilitates (1) fine-grained on-demand data transfer of remote images; (2) online decompression with efficient codecs; (3) trace-based prefetching; (4) peer-to-peer transfer to handle burst workload; (5) flexible choice of guest file systems and host systems; (6) efficient modification of large files (cross layer block references); (7) easy integration with the container ecosystem.

We have applied DADI to both cgroups [3] runtime and QEMU runtime. Support for other runtimes such as Firecracker [13], gVisor [5], OS^v [25], etc. should be technically straightforward. DADI has been deployed at scale in the production environment of Alibaba to serve one of the world's largest ecommerce platforms. Performance results show that DADI can cold start 10,000 containers on 1,000 hosts within 4 seconds. We are currently working on an edition of DADI for our public cloud service.

2 Background and Related Work

2.1 Container Image

Container images are composed of multiple incremental layers so as to enable incremental image distribution. Each layer is essentially a tarball of differences (addition, deletion or update of files) from a previous layer. The container system may apply the diffs in a way defined by its storage driver. The layers are usually much lighter than VM images that contain full data. Common layers are downloaded only once on a host, and are shared by multiple containers as needed. Each container has a dedicated writable layer (also known as container layer) that stores a private diff to the image, as shown in Figure 1. Writing to a file in the image may trigger a copy-on-write (CoW) operation to copy the entire file to the writable layer.

To provide a root file system to containers, the container engine usually depends on a union file system such as overlayfs, aufs, etc. These union file systems provide a merged view of the layers which are stored physically in different directories. The container system can also make use of Logical Volume Manager (LVM) thin-provisioned volumes, with each layer mapped to a snapshot.

The container system has a standard web service for image uploading and downloading called the Container Registry. The Container Registry serves images with an HTTP(S)-based protocol which, together with the incremental nature of layers, makes it a lot easier to distribute container images widely as compared to VM images.

2.2 Remote Image

The image distribution operation, however, consumes a lot of network and file system resources, and may easily saturate the service capacity allocated to the user/tenant, especially when creating or updating large container clusters. The result is long startup latencies for containers. After an image layer is received, it has to be unpacked. This unpacking operation is CPU, memory (for page cache) and I/O intensive at the same time so that it often affects other containers on the host and sometimes even stalls them.

To some extent, the current container image service is a regression to a decade ago when VM images were also downloaded to hosts. A similar problem has been solved once with distributed block stores [26, 28, 29, 38] where images are stored on remote servers, and image data is fetched over the network on-demand in a fine-grained manner rather than downloaded as a whole. This model is referred to as "remote image". There are several calls for this model in the container world (e.g. [18, 21]).

The rationale for remote image is that only part of the image is actually needed during the typical life-cycle of a container, and the part needed during the startup stage is even smaller. According to [19], as little as 6.4% of the image is used during the startup stage. Thus remote image saves a lot of time and resources by not staging the entire image in advance. And with the help of data prefetching (by OS) or asynchronous data loading (by applications themselves), the perceived time to start from a remote image can be effectively reduced further.

Remote image, however, requires random read access to the contents of the layers. But the standard layer tarball was designed for sequential reading and unpacking, and does not support random reading. Thus the format has to be changed.

2.3 File-System-Based Remote Image

CRFS [21] is a read-only file system that can mount a container image directly from a Container Registry. CRFS introduces an improved format called Stargz that supports random reads. Stargz is a valid tar.gz format but existing images need to be converted to realize remote image service. Instead of having the read-only file system read files directly from the layer, one could also extract the files in each layer and store them in a repository such as CernVM-FS [18] where they can be accessed on demand. CFS [27] is a distributed file system to serve unpacked layer files for hosts. Wharf [41], Slacker [19], Teleport [9] serve unpacked layer files through NFS or CIFS/SMB.

Due to the complexity of file system semantics, there are several challenges with file-system based image service. For example, passing a file system from host to guest across the virtualization boundary tends to limit performance. The I/O stack involves several complex pieces (including virtio-fs [10], FUSE [36], overlayfs [8], remote image itself) that need to be made robust and optimized. When compared to a block device, the file system also presents a larger attack surface that potentially reduces security in public clouds.

POSIX-compliant features are also a burden for non-POSIX workloads such as serverless applications, and an obstacle to Windows workloads running on Linux hosts (with virtualization). In addition, unikernel [5, 25, 39] based applications are usually highly specialized, and tend to prefer a minimalistic file system such as FAT [4] for efficiency. Some of them may even require a read-only file system. These different requirements are difficult to be satisfied by a file system that is predefined by the image service.

Furthermore, some desirable features of popular file systems such as XFS [11], Btrfs [2], ZFS [12], etc., are missing in current file-system-based image services, and are not likely to be supported soon. These include file-level or directory-level snapshot, deduplication, online defragmentation, etc. It is even difficult to efficiently (without copy) support standard features such as hard links and modification of files or file attributes.

2.4 Block-Snapshot-Based Remote Image

Modern block stores [26, 28, 29, 38] usually have a concept of copy-on-write snapshot which is similar to layer in the container world. Cider [16] and Slacker [19] are attempts to make use of such similarity by mapping image layers to the snapshots of Ceph and VMstore [17], respectively.

Container image layer and block store snapshot, however, are not identical concepts. Snapshot is a point-in-time view of a disk. Its implementation tends to be specific to the block store. In many systems, snapshots belong to disks. When a disk is deleted, its snapshots are deleted as well. Although this is not absolutely necessary, many block stores behave this way by design. Layer, on the other hand, refers to the incremental change relative to a state which can be that of a different image. Layer emphasizes sharing among images, even those belonging to different users, and has a standard format to facilitate wide distribution.

2.5 Others

File System Changesets. Exo-clones [33] implement volume clones efficiently with file system changesets that can be exported. DADI images are conceptually exo-clones with block level deltas that are not tied to any specific file system.

P2P downloading. Several systems allow container hosts to download image layers in a P2P manner, significantly reducing the download time in large environments [20, 22, 24, 30, 37]. VMThunder [40] adopts a tree-structured P2P overlay network to deliver fine-grained data blocks on-demand for large VM clusters. We reuse this general idea in DADI's optional P2P subsystem with a refined design and a production-level implementation.

Trimmed images. In order to pull less data and start a container in less time, DockerSlim [6] uses a combination of static and dynamic analyses to generate smaller-sized container images in which only files needed by the core application are included. Cntr [35] improves this by allowing dynamic accesses to trimmed files in uncommon cases via a FUSE-based virtual files system.

Storage Configuration for Containers. The layering feature of container image introduces new complexities in configuring storage. [34] demonstrates the impact of Docker storage configuration on performance.

VM images. Standard VM image formats such as qcow2, vmdk, vhd, etc. are block-level image formats and are technically reusable for containers. The major drawback of these image formats is that they are not layered. It is possible to emulate layering by repeatedly applying QEMU's backing-file feature, but doing this incurs significant performance overhead for reads. As we shall see in Section 3.1, the translation tables for standard VM image formats are also much bigger than those needed for DADI.

3 DADI Image Service

DADI is designed to be a general solution that can become part of the container ecosystem. The core of DADI (Sections 3.1-3.4) is a remote image design that inherits the layering model of container image, and remains compatible with the Registry by conforming to the OCI-Artifacts [31] standard. DADI is independent of transfer protocols so it is possible to insert an optional P2P transfer module to cope with large-scale applications (Section 3.5). DADI is also independent of the underlying storage system so users can choose an appropriate storage system such as HDFS, NFS, CIFS, etc., to form a fully networked solution (Section 4.4). DADI uses a block-level interface which minimizes attack surface, a design point especially relevant for virtualized secure containers.

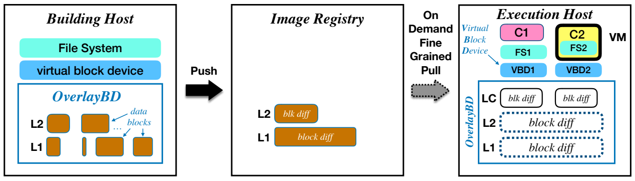


Figure 2: DADI Image. DADI image layer (L1, L2) consists of modified data blocks. DADI uses an overlay block device to provide each container (C1, C2) with a merged view of its layers.

3.1 DADI Image

As shown in Figure 2, DADI models an image as a virtual block device on which is laid a regular file system such as ext4. Note that there is no concept of file at the block level. The file system is a higher level of abstraction atop the DADI image. When a guest application reads a file, the request is first handled by the regular file system which translates the request into one or more reads of the virtual block device. The block read request is forwarded to a DADI module in user space, and then translated into one or more random reads of the layers.

DADI models an image as a virtual block device while retaining the layered feature. Each DADI layer is a collection of modified data blocks under the filesystem and corresponding to the files added, modified or deleted by the layer. DADI provides the container engine with merged views of the layers by an overlay block device (OverlayBD) module. We will use layer and changeset interchangeably in the rest of the paper for clearer statements. The block size (granularity) for reading and writing is 512 bytes in DADI, similar to real block devices. The rule for overlaying changesets is simple: for any block, the latest change takes effect. The blocks that are not changed (written) in any layer are treated as all-zero blocks.

The raw data written by the user, together with an index to the raw data, constitutes the layer blob. The DADI layer blob format further includes a header and a trailer. To reduce memory footprint and increase deployment density, we design an index based on variable-length segments, as illustrated in Figure 3. A segment tells where a change begins and ends in the image's logical block address (LBA) space, and also where the latest data is stored in the layer blob file's offset space. In this design, adjacent segments that are contiguous can be merged into a single larger segment to reduce the index size. The segment struct can record a change as small as 1 block which is the minimal write size for a block device. This avoids Copy-on-Write operations and helps to yield consistent write performance.

```
struct Segment {
  uint64_t offset:48;  // offset in image's LBA
  uint16_t length;     // length of the change
  uint64_t moffset:48; // mapped offset in layer blob
  uint16_t pos:12;     // position in the layer stack
  uint8_t flags:4;     // zeroed? etc.
  uint64_t end() {return offset + length;}
};
```

Figure 3: Definition of Segment. LBA is short for logical block address, offsets and lengths are in unit of blocks (512 bytes), size of the struct is 16 bytes.

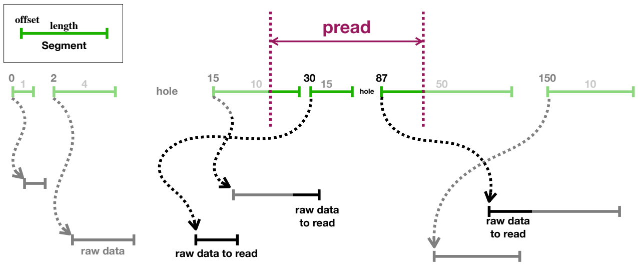


Figure 4: Index Lookup for Reading DADI Image. The lookup operation is a range query on a set of ordered non-overlapping variable-length segments, each of which points to the location of its raw data in the layer blob(s).

The index is an array of non-overlapping segments sorted by their offset. According to statistics from our production environment, the indices have fewer than 4.5K segments (see Section 5 for details) which corresponds to only 72KB of memory. In contrast, the qcow2 image format of QEMU has a fixed block size of 64KB by default, and an index based on radix-tree. QEMU allocates MBs of memory by default to cache the hottest part of its index.

To realize reading, DADI performs a range lookup in the index to find out where in the blob to read. The problem can be formally stated as given a set of disjoint segments in the LBA space, find all the segments (and "holes") within the range to read. This problem is depicted in Figure 4. For efficiency, the algorithm deals with variable-length segments directly without expanding them to fixed-sized blocks. As the index is ordered and read-only, we simply use binary search for efficient lookup, as shown in Algorithm 1. A B-tree could achieve higher efficiency but as the index contains only a few thousand entries in practice, we leave this optimization as a possible future work.

3.2 Merged View of Layers

When there are multiple layers, if the lookup procedure goes through the layers one by one, the time complexity is $O(n \cdot \log m)$ where n is the number of layers and m is the average number of segments in a layer. In other words, the cost increases linearly with n. We optimize this problem with a merged index that is pre-calculated when the indices are

```
Input: the range (offset, length) to look up
end ← offset + length;
i ← index.binary_search_first_not_less(offset);
if i < index.size() then
    delta ← offset - index[i].offset;
    if delta > 0 then // trim & yield 1st segment
        s ← index[i]; s.offset ← offset;
        s.moffset += delta; s.length -= delta;
        yield s; offset ← s.end(); i++;
    end
end
while i < index.size() and index[i].offset < end do
    len ← index[i].offset - offset;
    if len > 0 then // yield a hole
        yield Hole(offset, len);
        offset ← index[i].offset;
    end
    s ← index[i]; // yield next segment
    s.length ← min(s.length, end - offset);
    yield s; offset ← s.end(); i++;
end
if offset < end then // yield final hole
    yield Hole(offset, end - begin);
end
```

Algorithm 1: Index Lookup. Yields a collection of segments within the specified range (offset, length) with `i` initialized to the first element in the index that is not less than `offset`, and Hole being a special type of segment representing a range that has never been written.

```
Input: an array of indices[1..n];
       subscript i of the indices array for this recursion;
       the range to merge (offset, length)
for s in indices[i].lookup(offset, length) do
    if s is NOT a Hole then
        s.pos ← i;
        yield s;
    else if i > 0 then // ignore a real hole
        indices_merge(indices, i-1, s.offset, s.length);
    end
end
```

Algorithm 2: Index Merge by Recursion.

```
Input: an array of file objects blobs[0..n];
       a rage (offset, length) to pread
for s in merged_index.lookup(offset, length) do
    // s.pos == 0 for Hole segments
    // blobs[0] is a special virtual file object
    // that yields zeroed content when pread
    blobs[s.pos].pread(s.offset, s.length);
end
```

Algorithm 3: Read Based on Merged Index.

loaded, thus reducing the complexity to $O(\log M)$ where M is the number of segments in the merged index. The merging problem is illustrated in Figure 5.

To merge the indices, we put them in an array indexed from 1 to n where n is the number of layers, and in an order such that base layers come earlier. Algorithm 2 shows the recursive procedure to merge indices for a specified range. To merge them as whole, the algorithm is invoked for the entire range of the image. We make use of the `pos` field in the final merged index to indicate which layer a segment comes from. With the merged index, random read operation (pread) can be easily implemented as Algorithm 3, supposing that we have an array of file objects representing the ordered layers' blobs.

We analyzed 1,664 DADI image layers from 205 core applications in our production environment to extract the size of the merged indices. The statistics are summarized in Figure 6. They show that the indices have no more than 4.5K segments so the algorithm for merging indices is efficient enough to be run when an image is launched. Observe also that the number of segments is not correlated with the number of layers. This suggests that the performance of DADI OverlayBD does not degrade as the number of layers increases. Figure 7 plots the throughput of index queries on a single CPU core. Observe that at an index size of 4.5K segments, a single CPU core can perform more than 6 million index queries per second. In Section 5.4, we find that IOPS tops out at just under 120K for both LVM and DADI, suggesting that DADI spends no more than 1/50 of a CPU core performing index lookups.

3.3 Compression and Online Decompression

Standard compression file formats such as gz, bz2, xz, etc., do not support efficient random read operation. Files in these formats usually need to be decompressed from the very beginning until the specified part is reached. To support compression of the layers' blobs and enable remote image at the same time, DADI introduces a new compression file format called ZFile.

ZFile includes the source file compressed in a fixed-sized chunk-by-chunk manner and a compressed index. To read

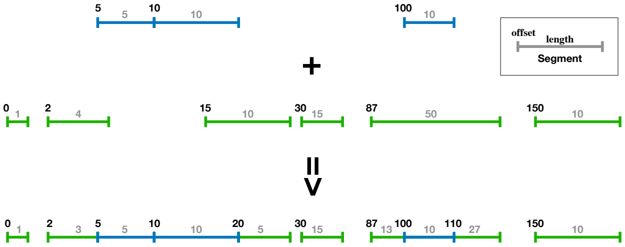


Figure 5: Index Merge.

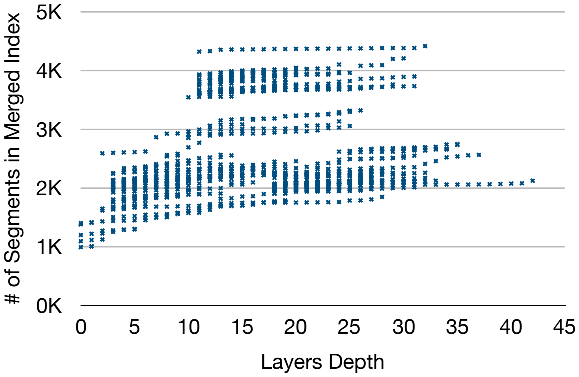


Figure 6: Index Size of Production Applications.

an offset into a ZFile, one looks up the index to find the offset and length of the corresponding compressed chunk(s), and decompresses only these chunks. ZFile supports various efficient compression algorithms including lz4, zstd, gzip, etc., and can additionally store a dictionary to assist some compression algorithms to achieve higher compression ratio and efficiency. Figure 8 illustrates the format of ZFile.

The index stored in ZFile is an array of 32-bit integers, each of which denotes the size of the corresponding compressed chunk. The index is compressed with the same compression algorithm as the data chunks. When loaded into memory, the index is decompressed and accumulated into an array of 64-bit integers denoting the offsets of the compressed chunks in the ZFile blob. After the conversion, index lookup becomes a simple array addressing at $offset/chunk\_size$.

Due to the fixed-size nature of chunks and the aligned nature of the underlying storage device, ZFile may read and decompress more data than requested by a user read. The decompression itself is an extra cost compared to the conventional I/O stack. In practice, however, ZFile improves user-perceived I/O performance even on servers with high-speed NVMe SSD. The advantage is even larger for slower storage (e.g. HDD or Registry). This is because, with the compression agorithm in use (lz4), the time saved reading the smaller amount of compressed data more than offsets the time spent decompressing the data. See Section 5.4 for detailed results.

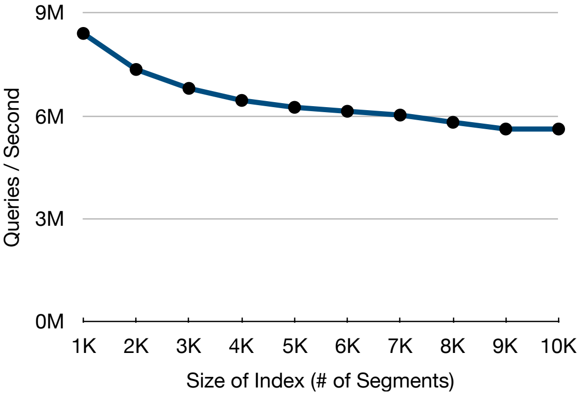


Figure 7: Index Performance on Single CPU Core.

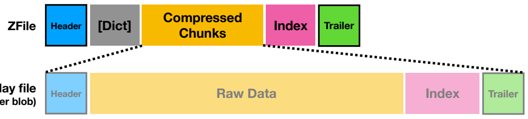


Figure 8: ZFile Format.

In order to support online decompression, a fast compression algorithm can be used at some expense to the compression ratio. We typically use lz4 in our deployment. Individually compressing chunks of the original files also impacts the compression ratio. As a result, DADI images are usually larger than the corresponding .tgz images but not by much.

We analyzed the blob sizes of 205 core applications in our production environment. Figure 9 shows the blob sizes in various formats relative to their .tar format sizes. In general, DADI uncompressed format (.dadi) produces larger blobs than .tar, due to the overhead of the image file system (ext4 in this case) but the overhead is usually less than 5% for layers that are larger than 10MB. Note that the compression ratio varies greatly among these images and some of them are not compressible. As discussed, ZFile blobs tend to be larger than their .tgz counterparts.

By adhering to the layered model of container image, DADI images are able to share layers. To further save space and network traffic, deduplication can be performed at the chunk level of DADI images, followed by compression of the unique chunks.

3.4 DADI Container Layer

Unlike other remote image systems (e.g. [18, 21]), DADI realizes a writable container layer. The writable layer is not only a convenient way to build new image layers, but also

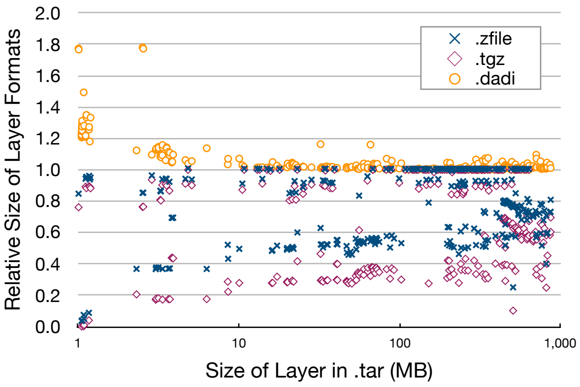


Figure 9: Relative Layer Blob Size.

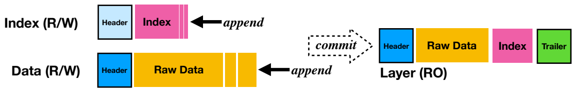


Figure 10: DADI's Writable Layer.

provides an option to eliminate the dependency on a union file system. We base the writable layer on a log-structured [32] design because this makes DADI viable on top of virtually all kinds of storage systems, including those that do not support random writes (e.g. HDFS). The log-structured writable layer is also technically a natural extension of the read-only layers.

As shown in Figure 10, the writable layer consists of one file for raw data and one for index. Both of these files are open-ended, and are ready to accept appends. As overwrites occur in the writable layer, these files will contain garbage data and index records. When there is too much garbage, DADI will spawn a background thread to collect the garbage by copying the live data to a new file, and then deleting the old file. When the writable layer is committed, DADI will copy the live data blocks and index records to a new file in layer format, sorting and possibly combining them according to their LBAs.

The index for the writable layer is maintained in memory as a red-black tree to efficiently support lookup, insertion and deletion. On a write, DADI adds a new record to the index of the writable layer. On a read, DADI first looks up the index of the writable layer. For each hole (a segment with no data written) within the range to read, DADI further looks up the merged index of the underlying read-only layers. DADI supports `TRIM` by adding to the writable layer an index record that is flagged to indicate that the range contains all-zero content.

3.5 P2P Data Transfer

Although remote image can greatly reduce the amount of image data that has to be transferred, there are situations where more improvement is necessary. In particular, there are several critical applications in our production environment that are deployed on thousands of servers, and that comprise layers as large as several GBs. The deployment of these applications places huge pressure on the the Registry and the network infrastructure.

To better handle such large applications, DADI caches recently used data blocks on the local disk(s) of each host. DADI also has the option to transfer data directly among the hosts in a peer-to-peer manner. Given that all the peers need roughly the same set of data and in roughly the same order during the startup time period, DADI adopts a tree-structured overlay topology to realize application-level multicast similar to VMThunder [40] instead of the rarest-first policy commonly used in P2P downloading tools [15, 20, 22].

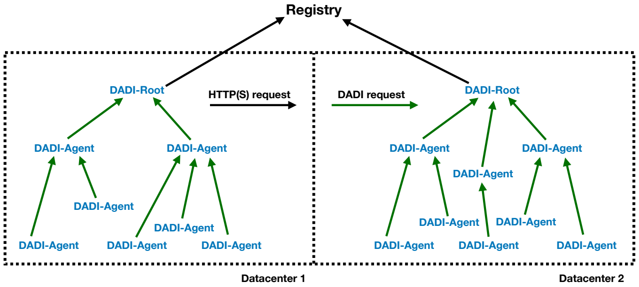


Figure 11: DADI's Tree-Structured P2P Data Transfer.

As shown in Figure 11, each container host runs a P2P module called DADI-Agent. In addition, there is a P2P module called DADI-Root in every data center that plays the role of root for topology trees. DADI-Root is responsible for fetching data blocks from the Registry to a local persistent cache, and also for managing the topology trees within its own datacenter. A separate tree is created and maintained for each layer blob.

Whenever an agent wants to read some data from a blob for the first time or when its parent node does not respond, it sends a request RPC to the root. The root may service the request by itself, or it may choose to rearrange the topology and redirect the requesting agent to a selected parent. The requesting agent is considered to join the tree as a child of the selected parent. Every node in a tree, including the root, serves at most a few direct children. If the requested data is not present in the parent's cache, the request flows upward until a parent has the data in its cache. The data received from the parent is added to the child's cache as it will probably be needed soon by other children or the node itself.

DADI-Root manages the topology. It knows how many children every node has. When a node needs to be inserted into the tree, the root simply walks down the tree in memory, always choosing a child with the fewest children. The walk stops at the first node with fewer direct children than a threshold. This node becomes the selected parent for the requesting agent. When a node finds that its parent has failed, it reverts to the root to arrange another parent for it. As the P2P transfer is designed to support the startup of containers and this startup process usually does not last long, DADI-Root expires topology information relatively quickly, by default after 20 minutes.

DADI-Root is actually a replicated service running on several servers for availability, and deployed separately for different clusters. An agent randomly chooses a root server in the same cluster when joining a transfer topology. It switches to another root server when it encounters a failure. The Registry tends to be shared by many clusters and possibly across a long distance so its performance may not always be high. In order to ensure that data blocks are likely to exist on the root when they are needed, we warm up the root servers' cache in

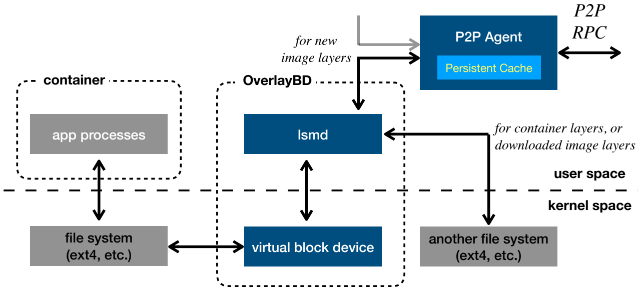


Figure 12: I/O Path for cgroups Runtime.

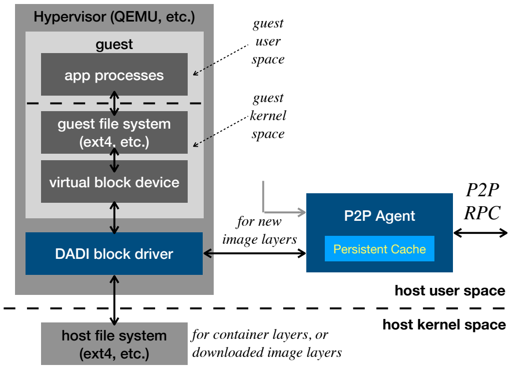


Figure 13: I/O Path for Virtualized Runtime (QEMU, etc).

our production environment whenever a new layer is built or converted.

To protect against potential data corruption, we create a separate checksum file for each and every layer blob as part of the image building or conversion process. The checksum file contains the CRC32 value for each fixed-sized block of the layer. As the checksum files are small, they are distributed whole to every involved node as part of the image pulling process. The data blocks are verified on arrival at each node.

4 Implementation and Deployment

This section discusses how DADI interfaces with applications and container engines, as well as how DADI can be deployed in different user scenarios.

4.1 Data Path

DADI connects with applications through a file system mounted on a virtual block device. DADI is agnostic to the choice of file system so users can select one that best fits their needs. By allowing the dependency on the file system to be explicitly captured at image creation time, DADI can help applications exploit the advanced features of file systems such as XFS [11], Btrfs [2], ZFS [12].

In the case of the cgroups runtime, we use an internal module called `vrbd` to provide the virtual block device. `vrbd` is similar to `nbd` but contains improvements that enable it to perform better and handle crashes of the user-space daemon. As shown in Figure 12, I/O requests go from applications to a regular file system such as ext4. From there they go to the virtual block device and then to a user-space daemon called `lsmd`. Reads of data blocks belonging to layers that have already been downloaded are directed to the local file system where the layers are stored. Other read operations are directed to DADI's P2P agent which maintains a persistent cache of recently used data blocks. Write and trim operations are handled by lsmd which writes the data and index files of the writable layer to the local file system.

We have also realized a QEMU driver for its block device backend to export an image to virtualized containers. As shown in Figure 13, the data path in this case is conceptually similar to that for the cgroups runtime except that the image file system and virtual block device are running in the guest context, and the block driver takes the place of lsmd. Integration with other hypervisors should be straightforward. It is also possible to pass the virtual block device from the host into the guest context. This approach works with virtually all hypervisors but incurs a slightly higher overhead. As the block device interface is narrower and simpler than a file system interface, it exposes a small attack surface to the untrusted guest container.

4.2 Container Engine Integration

DADI is integrated with Docker through a graph driver which is a Docker plugin to compose the root file system from layers. The layers form a graph (actually tree) topology hence the name graph driver. DADI is also integrated with containerd through a snapshotter which provides functions similar to those of the graph driver. We will use the term "driver" to refer to either of them in the rest of the paper.

We implemented the drivers to recognize existing and DADI image formats. When they encounter .tgz image, they invoke existing drivers. When they come across DADI image, they perform DADI-specific actions. In this way, the container engine can support both types of images at the same time so that the deployment of DADI to a host does not require the eviction of existing .tgz based containers or images from that host. This enables us to use a canary approach to systematically roll out DADI across our complex production environment.

DADI currently fakes the image pulling process with a small tarball file consisting of DADI-specific metadata. The tarball is very small so that the image pull completes quickly. We are preparing a proposal to the container community for extensions to the image format representation to enable lazy image pulling and make the engine aware of remote images.

4.3 Image Building

DADI supports image building by providing a log-structured writable layer. The log-structured design converts all writes into sequential writes so that the build process with DADI is usually faster than that for regular .tgz images (see Section 5 for details). As DADI uses faster compression algorithms, the commit operation is faster with DADI than it is for regular .tgz images. DADI also avoids pulling entire base images and this saves time when building images on dedicated image building servers where the base images are usually not already local.

In order to build a new layer, DADI first prepares the base image file system by bringing up a virtual block device and mounting the file system on it. When the layer is committed, DADI unmounts the file system and brings down the device. These actions are repeated for each layer produced in a new image, adding up to a lot of time. According to the specification of the image building script (dockerfile), each line of action will produce a new layer. It is not uncommon to see tens of lines of actions in a dockerfile in our environment so a single build job may result in an image with many new layers. This design was supposed to improve the speed of layer downloading by increasing parallelism, but it may become unnecessary with remote image.

We optimized the DADI image build process by bringing the device up and down only once. The intermediate down-and-ups are replaced with a customized operation called stack-and-commit. As its name suggests, stack-and-commit first stacks a new writable layer on top of existing layers, and then commits the original writable layer in the background. This optimization significantly increases image building speed, especially on high-end servers with plenty of resources.

To convert an existing .tgz image into the DADI format, DADI proceeds from the lowest layer of the image to its highest layer. For each layer, DADI creates a new writable layer and unpacks the corresponding .tgz blob into the layer while handling whiteouts, a special file name pattern that indicates deletion of an existing file. If users want to build a DADI image from a .tgz base image, the base image layers must first be converted into the DADI format using this process.

Some container engines implicitly create a special init layer named as xxxxx-init between the container layer and its images layers. This init layer contains some directories and files that must always exist in containers (e.g. /proc, /dev, /sys). During commit, DADI merges this init layer with the container layer so as to keep the integrity of the image file system.

4.4 Deployment Options

The P2P data transfer capability of DADI is optional and targeted at users with large applications. Other users may prefer to use DADI with the layer blobs stored in a high-performance shared storage system as a compromise between fetching the layer blobs from the Registry and storing the layer blobs on every host. Similar solutions have been proposed in the community (e.g. Teleport [9], Wharf [41]). DADI further enhances these solutions by not requiring the layers to be unpacked and supporting alternative storage systems such as HDFS.

For users who do not wish to set up shared storage, DADI provides them with the option to fetch layer blobs on-demand from the Registry and cache the data blocks on local disk(s). This approach greatly reduces cold startup latencies by avoiding the transfer of data blocks that are not needed. If there is a startup I/O trace available when launching a new container instance, DADI can make use of the trace to prefetch the data blocks needed by the starting container, yielding a near-warm startup latency. The trace can be simply collected with `blktrace`, and replayed with `fio`. See Section 5.2 for details.

Users may also choose to use DADI by downloading the layer blobs to local disk(s). DADI layers do not need to be unpacked, saving a time-consuming sequential process needed for .tgz layers. Thus pulling DADI images is much faster. The downloading can be optionally offloaded to P2P tools such as [20,22,24,30,37]. We use this approach as a backup path in case our on-demand P2P transfer encounters any unexpected error.

5 Evaluation

In this section, we evaluate the performance and scalability of DADI Image Service.

5.1 Methodology

We compare the container startup latency with DADI to that with the standard tarball image, Slacker, CRFS, LVM (dm or device mapper), and P2P image download. We also analyze the I/O performance as observed by an application inside the container.

Slacker uses Tintri VMstore as its underlying storage system. We do not have access to such a system so we use LVM together with NFS as an approximation of Slacker (denoted as pseudo-Slacker). At the time of this writing, CRFS has not yet achieved its goal of realizing an internal overlayfs so we rely on the kernel implementation of overlayfs for the comparisons.

We generally use NVMe SSDs as local storage. We also emulate a low-speed disk by limiting IOPS to 2,000 and throughput to 100 MB/s. These are the performance characteristics of the most popular type of virtual disks on public clouds so we refer to such a disk as “cloud disk” in the rest of the paper. We use ZFile by default for DADI unless explicitly noted. Before starting a test, we drop the kernel page cache in the host and guest (if applicable) as well as the persistent cache of DADI.

The physical servers we use are all equipped with dual-way multi-core Xeon CPUs and 10GbE or higher-speed NICs. The

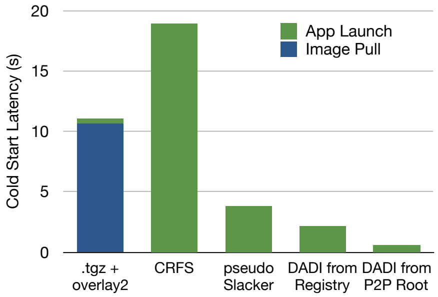


Figure 14: Cold Startup Latency.

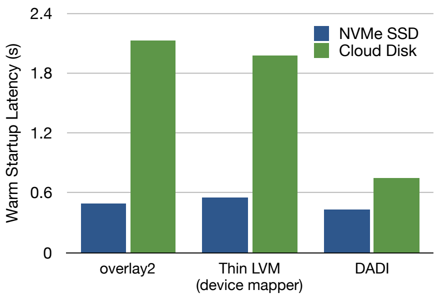


Figure 15: Warm Startup Latency.

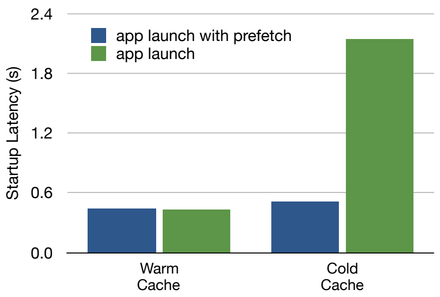


Figure 16: Startup Latency with Trace-Based Prefetch.

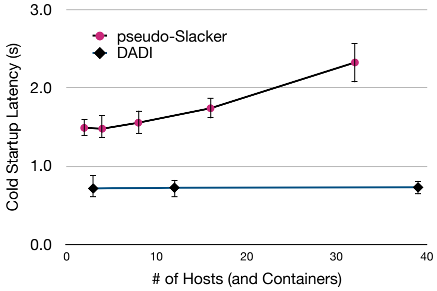


Figure 17: Batch Cold Startup Latency. Bars indicate 10 and 90 percentiles.

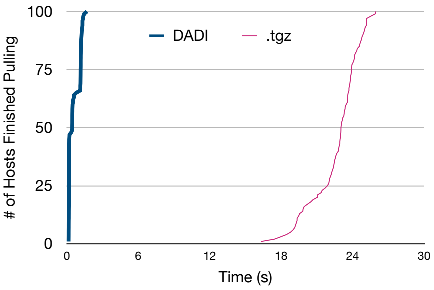


Figure 18: Time to Pull Image in Production Environment.

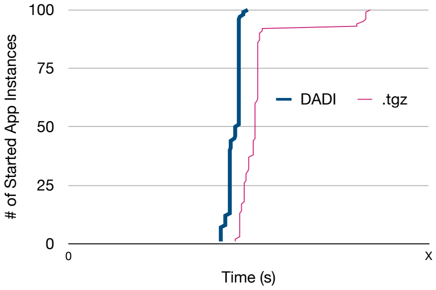


Figure 19: Time to Launch Application in Production Environment.

VMs are hosted on our public cloud. Each VM is equipped with 4 CPU cores and 8 GBs of memory. The vNICs are capable of a burst bandwidth of 5 Gbps and sustained bandwidth of 1.5 Gbps.

5.2 Startup Latency

To evaluate container startup latency, we use the application image `WordPress` from DockerHub.com. WordPress is the most popular content management system powering about one third of the Web in 2019 [7]. The image consists of 21 layers in .tgz format with a total size of 165MB. When unpacked, the image size is 501MB. In DADI compressed format with lz4 compression, the image occupies 274MB. The tarball of DADI-specific metadata that is downloaded on image pull is only 9KB in size.

Cold start of a single instance. We test startup latencies of a single container instance running WordPress when the layer blobs are stored in the Registry (.tgz, DADI, CRFS) and on remote storage servers (DADI, pseudo-Slacker). All the servers are located in the same datacenter as the container host. The results, as summarized in Figure 14, show that container cold startup time is markedly reduced with DADI.

Warm start of a single instance. Once the layer blobs are stored or cached on local disk, the containers can be started and run without a remote data source. In this case, any difference in startup time can be attributed to the relative efficiency of the I/O paths. As indicated in Figure 15, DADI performs 15%~25% better than overlayfs and LVM on NVMe SSD, and more than 2 times better on cloud disk.

Cold startup with trace-based prefetching. We first make use of `blktrace` to record an I/O trace when starting a container. On another host, we use `fio` to replay only the read operations in the trace while starting a new container instance of the same image. We set fio to replay with a relatively large I/O depth of 32 so as to fetch data blocks before they are actually read by the application. Figure 16 shows the results. Observe that trace-based prefetching can reduce 95% of the difference between cold and warm startup times.

Batch cold startup. In practice, many applications are large and require multiple instances to be started at the same time. For this batch startup scenario, we compare only pseudo-Slacker and DADI because the .tgz image and CRFS are bottlenecked by the Registry. The results are presented in Figure 17. Note that the startup time with pseudo-Slacker begins at 1.5s for one instance and increases to 2.3s for 32 instances. On the other hand, the startup time with DADI remains largely constant at 0.7s as the number of instances increases.

Startup in our Production Environment. We selected typical deployment tasks for an application in our production environment and analyzed its timing data. As shown in Figure 18, pulling the DADI metadata tarball takes no more than 0.2s for nearly half of the hosts and around 1s for the rest of the hosts. This compares very favorably with pulling the equivalent .tgz image which takes more than 20s for most

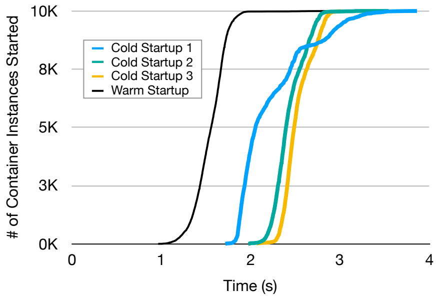


Figure 20: Startup Latency using DADI (Large-Scale Startup).

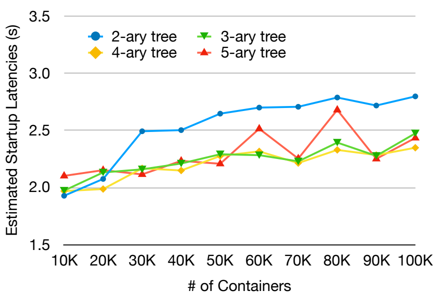


Figure 21: Projected Startup Latency using DADI (Hyper-Scale Startup).

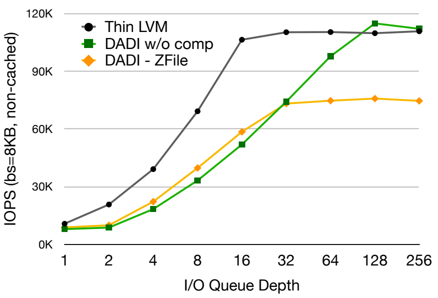


Figure 22: Uncached Random Read Performance.

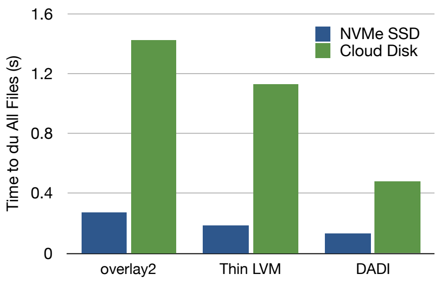


Figure 23: Time to `du` All Files.

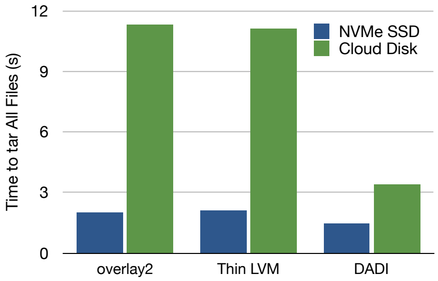


Figure 24: Time to `tar` All Files.

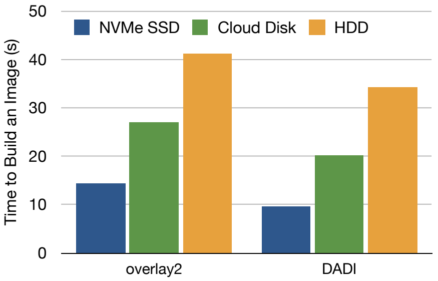


Figure 25: Time to Build Image.

of the hosts. Note that in this case, the .tgz image pull only needs to download the application layers as the much larger dependencies and OS layers already exist on the hosts. If all the layers have to be downloaded, the time needed will be even higher.

As shown in Figure 19, applications start faster using DADI remote image and P2P data transfer than with the .tgz image stored on local SSD. This result surprised us initially but it turned out to be a common occurrence for a couple of reasons. First, overlayBD performs better than OverlayFS (See Section 5.4). Second, with the tree-structured P2P data transfer, hosts effectively read from their parents' page cache, and this is faster than reading from their local disks.

5.3 Scalability

For the scalability analysis, we use a lightweight application called `Agility`. Agility is a Python application based on CentOS 7.6. Its image consists of 16 layers with a total size of 575MB in ZFile format and 894MB uncompressed. When Agility starts, it accesses a specified HTTP server which records the time stamps of all the accesses. We use Agility instead of WordPress for our scalability test because it provides a means to collect timings of a large number of container instances. Agility also consumes fewer resources, allowing us to create many more containers in our testbed.

Large-scale startup with DADI. We create 1,000 VMs on our public cloud platform and use them as hosts for containers. A large and increasing portion of our production environment is VM-based so this test reflects our real world situation. We start 10 containers running Agility on each host for a total of 10,000 containers. As shown in Figure 20, the cold start latency with DADI is within a second or two of that for warm start. The experimental environment is not dedicated and some noise is apparent in one of the runs (Cold Startup 1). Note that other than for ramp-up and long-tail effects, the time taken to start additional containers is relatively constant.

Hyper-scale startup with DADI. We deliberately construct a special P2P topology with tens of hosts and use it to project the behavior for a full tree with tens of thousands of hosts. The special topology models a single root-to-leaf path where each interior node has the maximum number of children. Each host again runs 10 instances of Agility. As shown in Figure 21, the startup time is largely flat as the number of containers increases to 100,000. Notice also that a binary tree for P2P is best when there are fewer than 20,000 participating hosts. A 3-ary or 4-ary tree works better beyond that scale.

5.4 I/O Performance

We perform micro benchmarks with `fio` to compare uncached random read performance. The results are summarized in Figure 22. At an I/O queue depth of 1, DADI offers comparable performance to LVM despite its user-space implementation. DADI's performance ramps up slower as the queue depth is increased but it catches up and tops LVM to achieve the highest IOPS at an I/O queue depth of 128 and without compression. This behavior suggests that DADI's index is more efficient than that of LVM, but there is room to optimize our queueing and batching implementation. Observe that DADI with compression performs 10%~20% better than without compression when the I/O queue depth is less than 32. This is because compression, by reducing the amount of data transferred, increases effective I/O throughput provided that the CPU is not bottlenecked. In our experimental setup, the CPU becomes bottlenecked for ZFile beyond a queue depth of 32.

We also test I/O performance with `du` and `tar` to scan the entire image from inside the container. These tests respectively emphasize small random read and large sequential read. The output of these commands are ignored by redirecting to `/dev/null`. As shown in Figure 23 and 24, DADI outperforms both overlayfs and LVM in all cases especially on the cloud disk. This is again primarily due to the effect of compression in reducing the amount of data transferred.

5.5 Image Building Speed

Image building speed is driven primarily by write performance and the time needed to setup an image. We evaluate image building performance with a typical dockerfile from our production environment. The dockerfile creates 15 new layers comprising 7,944 files with a total size of 545MB, and includes a few `chmod` operations that trigger copy-ups in overlayfs-backed images. As shown in Figure 25, the image is built 20%~40% faster on DADI than on overlayfs. Note that the time to commit or compress the image is not included in this measurement.

6 Discussion and Future Work

With overlayfs, containers that share layers are able to share the host page cache when they access the same files in those shared layers. Because DADI realizes each layered image as a separate virtual block device, when multiple containers access the same file in a shared layer, the accesses appear to the host to be for distinct pages. In other words, the host page cache is not shared, potentially reducing its efficiency.

One way to address this issue is to introduce a shared block pool for all the virtual block devices corresponding to the different containers on a host. The basic idea is to use the device mapper to map segments from the pool to the virtual block devices such that accesses by different containers to the same file in a shared layer appear to be for the same segment in the pool. The pool is backed by the page cache while the virtual block device and file system on top will need to support Direct Access (DAX) to avoid double caching. This solution can be further improved by performing block-level deduplication in the pool.

With the emergence of virtualized runtimes, container is becoming a new type of virtual machine and vice versa. The runtimes of container and VM may also begin to converge. By being based on the widely supported block device, DADI image is compatible with both containers and VMs, and is naturally a converged image service. Such a converged infrastructure will bring the convenience and efficiency of layered image to VM users on the cloud today. It will also provide users with increased flexibility and enable applications to evolve gradually from cloud-based to cloud-native.

A key part of realizing the potential of DADI is to standardize its image format and facilitate its adoption. We are working to contribute core parts of DADI to the container community.

7 Conclusions

We have designed and implemented DADI, a block-level remote image service for containers. DADI is based on the observation that incremental image can be realized with block-based layers where each layer corresponds to a set of file changes but is physically the set of changes at the block level underneath a given file system. Such a design allows the image service to be file system and platform agnostic, enabling applications to be elastically deployed in different environments. The relative simplicity of block-based layers further facilitates optimizations to increase agility. These include fine-grained on-demand data transfer of remote images, online decompression with efficient codecs, trace-based prefetching, peer-to-peer transfer to handle burst workload, easy integration with the container ecosystem. Our experience with DADI in the production environment of one of the world's largest ecommerce platforms show that DADI is very effective at increasing agility and elasticity in deploying applications.

Acknowledgments

This paper would not have been possible without the close collaboration of our storage, container, serverless, kernel and virtualization teams. We are especially grateful to Haobo Xu, Liang Han, Qianbo Huai and Jiwei Lu for delivering the initial version of the P2P Data Transfer capability. We are also grateful to our shepherd, Vasily Tarasov, and anonymous reviewers for helping us improve the paper.

References

[1] 9p virtio - KVM. https://www.linux-kvm.org/page/9p_virtio. Accessed: 2020-01-15.

[2] Btrfs. https://btrfs.wiki.kernel.org/index.php/Main_Page. Accessed: 2020-01-15.

[3] Everything You Need to Know about Linux Containers, Part I: Linux Control Groups and Process Isolation. https://www.linuxjournal.com/content/everything-you-need-know-about-

linux-containers-part-i-linux-control-groups-and-process. Accessed: 2020-01-15.

[4] File Allocation Table. https://en.wikipedia.org/wiki/File_Allocation_Table. Accessed: 2020-01-15.

[5] gVisor. https://gvisor.dev/. Accessed: 2020-01-15.

[6] Minify and Secure Your Docker Containers. Frictionless! https://dockersl.im/. Accessed: 2020-01-15.

[7] One-third of the web! https://wordpress.org/news/2019/03/one-third-of-the-web/. Accessed: 2020-01-15.

[8] Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed: 2020-01-15.

[9] Project Teleport. https://azure.microsoft.com/en-gb/resources/videos/azure-friday-how-to-expedite-container-startup-with-project-teleport-and-azure-container-registry/. Accessed: 2020-01-15.

[10] virtio-fs. https://virtio-fs.gitlab.io/. Accessed: 2020-06-05.

[11] XFS. http://xfs.org/. Accessed: 2020-01-15.

[12] ZFS: The Last Word in Filesystems. https://blogs.oracle.com/bonwick/zfs:-the-last-word-in-filesystems/. Accessed: 2020-01-15.

[13] Amazon. Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/. Accessed: 2020-01-15.

[14] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[15] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[16] Lian Du, Tianyu Wo, Renyu Yang, and Chunming Hu. Cider: A Rapid Docker Container Deployment System through Sharing Network Storage. In *2017 IEEE 19th International Conference on High Performance Computing and Communications (HPCC'17)*, pages 332–339. IEEE, 2017.

[17] Gideon Glass, Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo. Logical Synchronous Replication in the Tintri VMstore File System. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 295–308, 2018.

[18] N Hardi, J Blomer, G Ganis, and R Popescu. Making Containers Lazy with Docker and CernVM-FS. In *Journal of Physics: Conference Series*, volume 1085, page 032019. IOP Publishing, 2018.

[19] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 181–195, 2016.

[20] Alibaba Inc. Dragonfly: An Open-source P2P-based Image and File Distribution System. https://d7y.io/en-us/. Accessed: 2020-01-15.

[21] Google Inc. CRFS: Container Registry Filesystem. https://github.com/google/crfs. Accessed: 2020-01-15.

[22] Uber Inc. Introducing Kraken, an Open Source Peer-to-Peer Docker Registry. https://eng.uber.com/introducing-kraken/. Accessed: 2020-01-15.

[23] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[24] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. FID: A Faster Image Distribution System for Docker Platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 191–198. IEEE, 2017.

[25] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv–Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 61–72, 2014.

[26] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. URSA: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 15. ACM, 2019.

[27] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. CFS: A Distributed

File System for Large Scale Container Platforms. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1729–1742. ACM, 2019.

[28] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 257–273, 2014.

[29] Kazutaka Morita. Sheepdog: Distributed Storage System for QEMU/KVM. *LCA 2010 DS&R miniconf*, 2010.

[30] Aravind Narayanan. Tupperware: Containerized Deployment at Facebook, 2014.

[31] OCI. OCI Artifacts. `https://github.com/opencontainers/artifacts`. Accessed: 2020-01-15.

[32] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[33] Richard P Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Rawlinson Rivera, and Christos Karamanolis. Exo-clones: Better Container Runtime Image Management across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, 2016.

[34] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 199–206. IEEE, 2017.

[35] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. CNTR: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 199–212, 2018.

[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies FAST'17)*, pages 59–72, 2017.

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[38] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[39] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 173–186, 2018.

[40] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338, 2014.

[41] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 174–185. ACM, 2018.

Efficient Miss Ratio Curve Computation for Heterogeneous Content Popularity

Damiano Carra
University of Verona, Italy

Giovanni Neglia
Inria, Université Côte d'Azur, France

Abstract

The Miss Ratio Curve (MRC) represents a fundamental tool for cache performance profiling. Approximate methods based on sampling provide a low-complexity solution for MRC construction. Nevertheless, in this paper we show that, in case of content with a large variance in popularity, the approximate MRC may be highly sensitive to the set of sampled content. We study in detail the impact of content popularity heterogeneity on the accuracy of the approximate MRC. We observe that few, highly popular, items may cause large error at the head of the reconstructed MRC.

From these observations, we design a new approach for building an approximate MRC, where we combine an exact portion of the MRC with an approximate one built from samples. Results for different real-world traces show that our algorithm computes MRC with an error up to 10 times smaller than state-of-the-art methods based on sampling, with similar computational and space overhead.

1 Introduction

Caches have been widely used in different contexts to improve system performance, from CPU, to disk, to web. As the architectures become more complex, with multi-core CPUs, or clusters of machines, caches maintain a key role in providing fast access to the most used content. Being a shared resource, a cache may be misused by some aggressive processes or application, hurting the performance of other processes.

To provide a fair sharing, one may virtually divide the cache and assign dynamically different portions to specific applications or types of applications. For instance, Sundarrajan *et al.* [33] show that, in case of Web caches, video streaming, web browsing, and software updates have extremely different content access patterns and cache resource requirements. Similarly, when multiple VMs run on the same physical host, efficient sharing of storage resources, like a SSD used as cache, needs detailed VM profiling [21, 24, 28, 32]. Analogous observations have been made in different contexts, such as multiprocessor systems [10, 19] and distributed processing in datacenters [27, 35].

The most important performance metric for a cache is usually the *hit ratio*. For cache partitioning, it is necessary to quantify the hit ratio a given application would experience given the amount of available cache space. This relation is captured by the *Miss Ratio Curve* (MRC), which gives the miss ratio as a function of the cache size. The use of MRCs can be helpful also in contexts where caches can be provisioned *on demand* [7, 29] with a pay-as-you-go model, as it is in the case for cloud caches [1–3].

MRC can be computed analytically for many caching policies—sometimes exactly [14, 20], more often approximately [15, 16]—but only under idealized models for the request process. Real traffic usually exhibits complex patterns that diverge from these models.

A more common approach, dating back to Mattson's seminal work [25], is to compute the MRC directly from the trace of the specific workload. The MRC can be built with $\mathcal{O}(\log M)$ computational complexity per request, and $\mathcal{O}(M)$ memory [9, 13], where M is the number of distinct items that are requested. Since content popularity (and consequently the MRC) may vary over time, the usual approach is to select an interval of time over which the traffic request process may be considered stationary. Then, the requests observed during this interval are used to build a MRC, which drives the resource assignment for the next interval. In case of high traffic rate, if we need to continuously build many MRCs for different application types, computational complexity and memory requirements may represent a heavy burden [7, 33].

For this reason, by trading accuracy with computational complexity and memory, recent works propose to compute approximate MRCs with $\mathcal{O}(1)$ operations per request, and $\mathcal{O}(1)$ memory [17, 29, 36, 37]. Such low-complexity solutions are based on the common idea of sampling the trace.

Limitation of the prior work. Sampling has been applied widely in different domains. A potential pitfall of sampling is the introduction of biases. In building the approximate MRC, there could be two approaches: sampling the *requests* [4, 34], or sampling the *items* and observing the requests for those items [36, 37]. Request sampling introduces a bias [30, 39], which motivated the introduction of item sampling, also called *spatial sampling*. Nevertheless, if request rates vary greatly across items, spatial sampling can be biased too, a fact that was implicitly acknowledged by Waldspurger *et al.* [37]. To the best of our knowledge, we are the first to thoroughly address and explain such a bias in detail.

As an example, the left column of Figure 1 shows the exact and approximate MRCs using the LRU eviction policy, built from various samples, considering traces with different traffic characteristics. In particular, item request frequencies, usually referred to as *popularities*, are Zipf-distributed with two different values of the Zipf exponent (α)—experiments' details are provided in Section 3, but they are not essential to understand what follows. With higher values of α, the distribution becomes more skewed, and therefore popularities become highly heterogeneous. The approximate MRCs in the left column are obtained using SHARDS [37] with a constant sampling rate $R = 0.01$. The experiments show that SHARDS is able to obtain an accurate MRC when item popularity is not highly skewed. But, as heterogeneity increases, the error drastically increases. Waldspurger *et al.* [37] recognized this possible bias, and proposed the variant SHARDS$_{\text{adj}}$ that partially solves the problem. The curves in Figure 1, right column, show that SHARDS$_{\text{adj}}$ correctly estimates the tail of the MRC, but not its head, with large errors for high miss ratio values (above 30% in the bottom subfigure) that may even exceed 100% (top subfigure). Miss ratios above 70% are the norm for many caches, including HDD [37] and SSD ones [21, 24], and hierarchical web caches, where the higher level resides in RAM [5, 26]. Caches consist of fast, expensive storage, and they are inherently a scarce, shared resource. An accurate assignment requires the knowledge of the MRC for *any size*, even when the miss ratio is large.

Contributions. In this work we study the impact of heterogeneity on the accuracy of the approximate MRC. With the help of different sets of experiments, and a model of a representative scenario, we observe that highly popular item play a fundamental role, and we shed lights on the fact that the MRC is highly sensitive to the specific content sampled. Consequently, we design a new approach, where we combine exact MRC computation for small values of the cache capacity (which is mainly influenced by popular items) with approximate computation for larger values. We evaluate our scheme with both synthetic and real-world traces. Our results show that our method is able to reconstruct the MRC with an average error up to 10 times smaller than state of the art approaches, with the same complexity. We also consider a scenario where items have heterogeneous sizes, and show how our solution correctly addresses it.

Roadmap. The remaining of the paper is organized as follows. In Section 2 we provide the background information and discuss the related work. In Section 3 we study the impact of popular items on the accuracy of the spatial sampling approaches. Section 4 presents our solution, which is evaluated in Section 5. Section 6 provides additional considerations on the scheme we propose, and Section 7 concludes the paper.

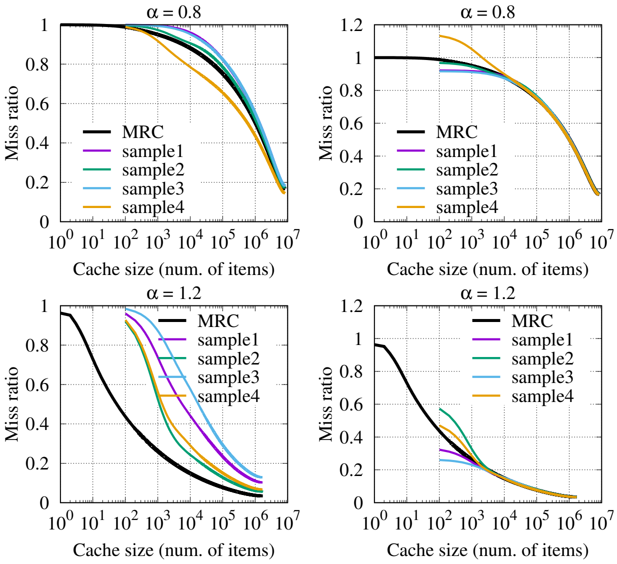


Figure 1: Approximate MRCs built from samples compared to the exact MRC for different values of the parameter α of the Zipf distribution used for item popularity ($R = 0.01$): SHARDS (left column) and SHARDS$_{\text{adj}}$ (right column).

2 Background and Related Work

The MRC can be computed with a single pass on the request trace if the eviction policy satisfies the inclusion property, *i.e.*, the set of items stored in the cache at a given time is a subset of the set of items that would be stored if the cache had a larger size [25]. Widely adopted policies such as LRU, LFU, and MRU satisfy such property, therefore MRCs are useful in many practical systems.

Methods with different computational and memory requirements have been proposed to build the MRC [6, 37]. We describe here a specific algorithm suitable when all items have the same size. The caching policies listed above all maintain an ordered list of the items in the cache, where, at any instant, the last item in the list is the current candidate to be evicted. The MRC algorithm goes through the trace, maintaining an ordered list $\mathcal{T}$ of references to the items mimicking how the corresponding caching policy would work if the cache size were infinite. Given a request for item j, if the item is not in the cache, we have a *cold miss*. If the item is in the cache, then the algorithm determines its current position (called the *reuse distance*) and updates an empirical histogram of reuse distances. Once the trace is analyzed, the histogram is normalized dividing each value by the total number of requests. By summing the histogram values up to a given capacity C, one obtains the corresponding hit ratio, whose one's complement is the miss ratio.

Exact MRC computation requires $\mathcal{O}(M)$ memory, where M is the number of distinct items in the trace, and has a computational complexity of $\mathcal{O}(\log M)$ per request due to the access to $\mathcal{T}$, which can be implemented with a tree data structure [40]. The approximate solutions based on sampling may adopt two approaches: request sampling and item sampling. The solutions based on request sampling—such as sampling every n requests [4], or sampling for small intervals of time [34]—are known to be biased [30, 39]. To overcome these issues, item sampling has been recently proposed for computing the reuse distance to characterize the use of storage memory [38] or program locality in single core [12] and multi-core architectures [30], and for building approximate MRC with low computational complexity [17, 36, 37].

In this work we consider the solution adopted by SHARDS [37]. SHARDS selects randomly a fraction R of the items, computes the MRC considering only the requests from these items, and then scales the cache capacity on the X-axis by a factor $1/R$. The item selection is done using a hash of the item identifier, id_j. Since sampling may exclude or include very popular items, the authors of SHARDS proposed an adjustment, called SHARDS$_{\text{adj}}$, in which the estimated miss ratios are scaled up by the ratio between the actual and the expected number of sampled references.

How to measure the Accuracy. In evaluating the approximate MRCs, accuracy is usually measured using the Mean Absolute Error (MAE): this is the average of the absolute differences between the exact and the approximate MRC for all cache sizes considered. Such a metric is easy to interpret, but it gives the same importance to all different values of cache size. The following simple example illustrates a potential problem. Figure 2 shows the exact and approximate MRC obtained with SHARDS for a publicly available trace which we name *ms-ex*—trace details in Section 5.3. In the figure on the left, the two curves look similar and indeed the MAE is only 0.025, *i.e.*, if we pick an arbitrary value of the cache size, on average the approximate MRC allow us to estimate the miss rate with an error of ±2.5%. The figure on the right contains the same information but using a log scale for the X-axis. It appears that the average error is not representative. In fact, the error for small cache sizes can be 5-6 times bigger.

In this paper we introduce a *new metric*—MAEQ, Mean Absolute Error per Quantile—that maintains the simplicity of MAE, but it takes into account how much the MRC varies in the different intervals. While the MAE provides the average error for a cache size sampled uniformly at random, the MAEQ provides the average error when a *miss ratio* is sampled uniformly at random. In particular, the metric is based on the concept of *quantiles*. We consider different uniformly spaced quantiles for the miss ratio and identify the corresponding cache size intervals. For instance, if we consider the quantiles at 0.8, 0.6, 0.4, and 0.2 in Figure 2, right, we identify the following intervals for cache sizes: The first interval goes from 0 to 50, the second one from 50 to 10^5, the third from 10^5 to $8 \cdot 10^5$, and the fourth one from $8 \cdot 10^5$ till the end. Note that no portion of the MRC falls below 0.2, so the last quantile is not considered. Once we have identified the ranges that correspond to the quantiles, we compute the MAE in each range, and then we average the MAE. More formally, $MAEQ = \sum_{i=1}^{Q^*} MAE_i/Q^*$, where the interval i is defined by the quantile, and Q^* is the number of quantile intervals considered. In practice, we consider quantiles with a step increment of 0.01: for each variation of 0.01 in the miss ratio, we compute the MAE, and then we take the average of the MAEs. If we compute the MAEQ for the above trace, we obtain 0.090, which provides a better idea of the accuracy of the approximate MRC, when we look at different miss ratio ranges.

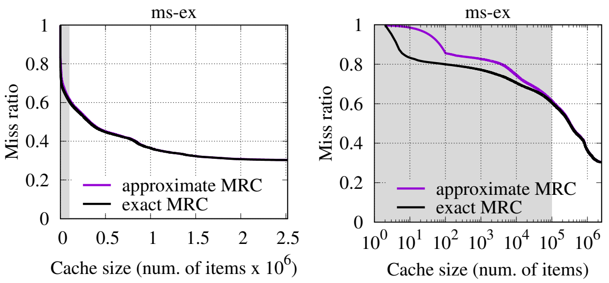


Figure 2: Error between approximate and exact MRCs (without and with log scale in the X-axis). The shaded area highlights the portion where the approximate and exact MRCs differ the most.

3 Evaluation of Spatial Sampling Approaches

3.1 Evaluation Methodology and Settings

In this section we evaluate the impact of content popularity heterogeneity on the accuracy of the approximate MRC. We consider the SHARDS and SHARDS$_{\text{adj}}$ approaches [37] with a fixed sampling rate, denoted by R, which varies from 0.1 to 0.001. In order to have highly controllable experiments, we first consider a set of traces generated according to the Independent Reference Model (IRM), in which the probability that the next request for item i is constant over time and independent from the previous requests. We call this probability the *popularity* of content i and denote it as p_i. In particular, we use the Zipf popularity distribution ($p_i \propto i^{-\alpha}$) with different values of the parameter α. We have tested different combinations of catalogue size and trace length; in what follows we report the representative results for the case with 50M requests for a set of 10M items and different values of the parameter α.

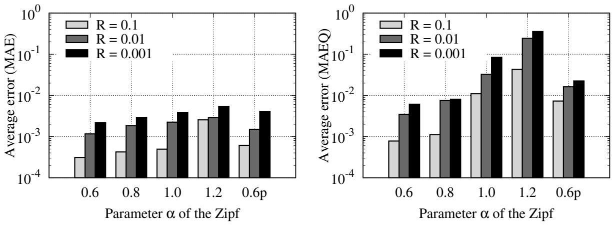


Figure 3: Accuracy for different values of the parameter α of the Zipf distribution used for item popularity, and for different sampling rates R: MAE (left) and MAEQ (right).

3.2 Results with the IRM Traces

Figure 1 shows the exact and approximate MRCs for different values of α (sampling rate $R = 0.01$). The results in left columns have been obtained using SHARDS, while the ones in the right column using SHARDS$_{\text{adj}}$. Note that, with spatial sampling, we are able to build the MRC at points that are multiples of $1/R$—this is why the approximated MRCs start at $1/R = 100$.

In case of SHARDS, as α increases, the error increases significantly. The problem is partially solved by SHARDS$_{\text{adj}}$, whose effect is to decrease the error in the tail of the MRC. This is obtained by rescaling the approximated MRC, but such a rescaling has an impact on the whole MRC, and it leads to significant errors for small cache values yielding miss ratios larger than 1. This detail was not discussed in the SHARDS work [37]. Our model in Section 3.3 explains these findings.

In Figure 3 we show SHARDS$_{\text{adj}}$ MAE and MAEQ values for different values of the sampling rate R and the Zipf parameter α—the case labeled as "0.6p" will be discussed later. Each value has been computed averaging five different samples. The MAE indicates an error smaller than 0.006, but, as we discussed above, such metric considers all cache sizes equally important. The MAEQ, instead, indicates an average error over the quantiles that, for $\alpha = 1.2$ and $R = 0.001$ may be as high as 0.35, which better describes the difference between the exact and the approximate MRC.

On the head of the MRC. The presence of a relatively small number of *highly popular items* determines the accuracy of MRCs. If we observe the empirical item popularity distribution in real-world traces—we will show some examples in Figure 9—we notice that there are often two groups of items: a small group with very popular items, and a large one with much less popular items.

Inspired by these real-world traces, we modify a Zipf distribution with $\alpha = 0.6$ by adding 20 popular items, whose popularity is randomly selected between 0.005 and 0.01. Popularities have been normalized to guarantee that their sum equals one. Figure 4, left, compares this new popularity distribution, labeled as "0.6p", with the Zipf distribution used in the previous section. Figure 4, center and right, shows approximate MRCs for a sampling rate $R = 0.01$ with SHARDS and SHARDS$_{\text{adj}}$, respectively. With the addition of a few popular items (20 out of 10M) the accuracy of the approximate MRC is significantly affected. The MAE and the MAEQ are shown in Figure 3, with the label "0.6p." This experiment confirms that, when sampling fails to capture the contribution of the popular items, the result may be heavily biased. On the other hand, less popular items have limited impact on the MRC and they may be sampled randomly.

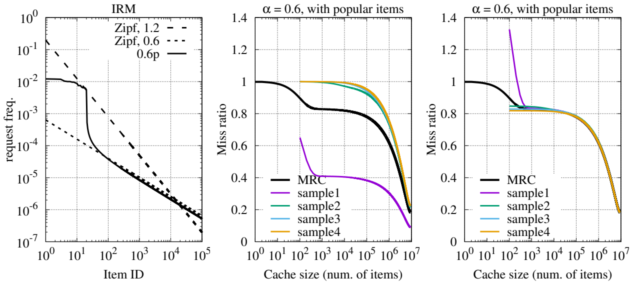


Figure 4: Distributions used for the experiments (left) and MRCs built from samples for the "0.6p" case ($R = 0.01$) through SHARDS (center) and SHARDS$_{\text{adj}}$ (right).

3.3 Understanding the role of popular items

We analyze a *simple scenario* for which we can derive an approximate model. Consider a finite set of items where there is a single very popular item (content c_1), with popularity p, and M items with approximately homogeneous popularity, *i.e.*, each item has a popularity of approximately $(1-p)/M$, and $p \gg (1-p)/M$. The request sequence is generated following the IRM model. Let r_n denote the n-th request. We consider the *reuse distance*, *i.e.*, the number of unique references between two references to the same item [40], and the *reuse time*, *i.e.*, the total number of references between two references to the same item [17].

For small cache sizes, the reuse distance can be approximated by the reuse time, *i.e.*, misses for content c_1 occur (almost) every time the reuse time for the content exceeds the cache size C.[1] The reuse distance is geometrically distributed with expected value $1/p$. Then, the miss probability for content c_1 starts decreasing significantly as the cache reaches size $1/p$, and decreases exponentially fast for bigger and bigger cache sizes. Once the cache size is 3 to 4 times larger than $1/p$, content c_1 is highly likely to be in the cache and the miss ratio is at most $1-p$. For larger cache sizes, the miss rate still decreases because of the contribution of the unpopular items. The decrease is now linear in the cache size. This reasoning can be made formal, *e.g.*, using a simple model based on the Che's approximation [11], and we obtain that the miss ratio

[1] This is an approximation because requests for other items contribute to move content c_1 closer to the tail only if they are misses. But, for small cache sizes, almost all requests for the M unpopular items generate misses.

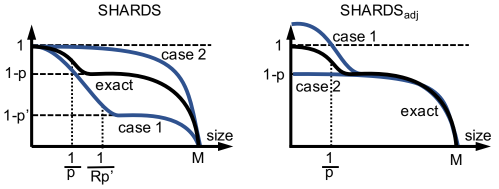


Figure 5: Simple scenario: exact MRC, along with two cases of approximate MRCs.

when the cache has capacity $C \in [0, M]$ is:

$$m(C) \approx pe^{-pC} + (1-p)\left(1 - \frac{C}{M}\right). \tag{1}$$

Figure 5 shows a sketch of what the miss ratio looks like using a logarithmic scale for the capacity axis—curve labeled "exact."

Now, assume we sample the items with sampling rate R. Either the set of sampled items contains the very popular content (with probability R), or it does not contain it (with probability $1-R$). In the following, we consider these two cases and show that the resulting MRCs differ significantly from the exact MRC.

Case 1: content c_1 is sampled. The sample contains requests for content c_1 and, on average, $M' = RM$ unpopular items. The fraction of requests for content c_1 is $p' = p/(p+(1-p)R) > p$. The exact MRC of the sampled trace is determined by (1) with M' and p' replacing respectively M and p. For the approximate MRC, we replace C with RC and we obtain:

$$m(C) \approx p'e^{-p'RC} + (1-p')\left(1 - \frac{C}{M}\right). \tag{2}$$

The curve labeled "case 1" in Figure 5, left, corresponds to (2). We observe a fast decrease of the miss ratio for cache sizes around $1/(p'R)$ from 1 to $1-p' < 1-p$. Then, the approximate MRC underestimates the miss ratio at least for large values of the cache size.

Case 2: content c_1 is not sampled. In this case there are on average $M' = RM$ equally popular items in the cache, then the miss ratio is

$$m(C) \approx 1 - \frac{C}{M}, \tag{3}$$

and is represented by the curve labeled "case 2" in Figure 5, left.

Adjustment proposed in SHARDS$_{\text{adj}}$. SHARDS$_{\text{adj}}$ rescales the estimated MRC by a factor $N_S/(RN)$, where N_S is the number of requests observed in the sample. For *Case 1*, we have $N_S = pN + RN(1-p)$, and we obtain

$$m_{\text{adj}}(C) \approx \frac{p}{R}e^{-p'RC} + (1-p)\left(1 - \frac{C}{M}\right). \tag{4}$$

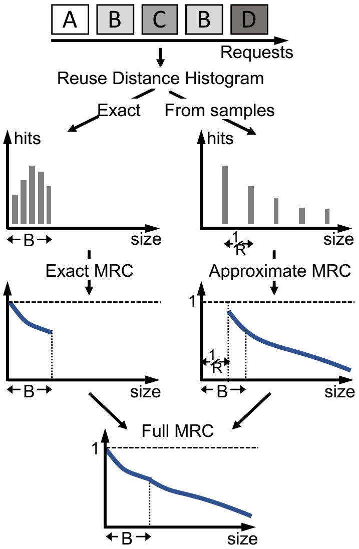


Figure 6: Approximate MRC building process.

We observe then how SHARDS$_{\text{adj}}$ removes the bias for large capacity values but amplifies it for small ones, leading to a miss ration greater than 1 in this example ($m_{\text{adj}}(0^+) \approx p/R + (1-p) > 1$). Similarly, for *Case 2*, we obtain

$$m_{\text{adj}}(C) \approx (1-p)\left(1 - \frac{C}{M}\right). \tag{5}$$

SHARDS$_{\text{adj}}$ correctly predicts the tail of the MRC, as for *Case 1*, but it now underestimates the head. Figure 5, right, shows the curves that corresponds to *Case 1* and *Case 2* when using SHARDS$_{\text{adj}}$.

4 Proposed solution

In the previous section we observed that popular and less popular items have different impacts on the MRC building process. These observations motivate our solution. We design a scheme, where we combine an exact MRC for small cache sizes (where the miss ratio depends mostly on the highly popular items), with an approximate MRC built from sampled items for large cache sizes. In particular, for the approximate MRC we use SHARDS$_{\text{adj}}$, because it predicts the MRC tail better. The approach is qualitatively illustrated in Figure 6. For the portion of the MRC built from samples, the general scheme can adopt either a *constant sampling rate* or an adaptive sampling rate to achieve *constant computational complexity*. We discuss these two schemes below.

Constant Sampling Rate Scheme. Algorithm 1 describes the solution we propose. We assume the use of the LRU eviction policy, but the scheme may be adapted easily to any policy that satisfies the inclusion property. The algorithm requires two parameters: B, which is the number of items for

Algorithm 1: Approximate MRC building process

```
input : $B$, number of positions for the exact MRC
input : $R_s$, sampling rate for the approximate MRC
input : request sequence
$\mathcal{T}_e \leftarrow$ countingBTree(); $V_e \leftarrow$ reuseVector();
$\mathcal{T}_s \leftarrow$ countingBTree(); $V_s \leftarrow$ reuseVector();
foreach request $r$ for item with id $j$ do
    if $(j \in \mathcal{T}_e)$ then
        $\text{pos}_j \leftarrow \text{remove}(j, \mathcal{T}_e)$;
        update $V_e$ at $\text{pos}_j$;
    add $j$ to $\mathcal{T}_e$;
    if $(size(\mathcal{T}_e) > B)$ then
        remove last item from $\mathcal{T}_e$;
    if $(hash(j) \bmod P < R_s P)$ then
        // sampled item
        if $j \in \mathcal{T}_s$ then
            $\text{pos}_j \leftarrow \text{remove}(j, \mathcal{T}_s)$;
            update $V_s$ at $\text{pos}_j$;
        add $j$ to $\mathcal{T}_s$;
MRC$[0..B] \leftarrow$ buildExactMRC$(V_e)$;
MRC$[B..\infty] \leftarrow$ buildApproxMRC$(V_s)$;
```

the exact MRC, and R_s, which is the sampling rate. Given a set of requests for an unknown catalogue of items, we maintain two tree data structures respectively to build the exact MRC ($\mathcal{T}_e$), and the approximate one ($\mathcal{T}_s$). $\mathcal{T}_e$ size equals B item references, while $\mathcal{T}_s$ depends on the number of items in the trace and the sampling rate. *Splay Trees* or *Counting BTrees* are possible candidates for such data structures, since they have logarithmic complexity for the insert/delete operations. Instead, for the lookup we maintain a hash table.

Once the trace is processed, we build the exact MRC ($m_e(C)$) and the approximate one ($m_s(C)$). We then connect by continuity the approximate MRC starting from B, and modulating exponentially any potential step discontinuity (equal to $m_e(B) - m_s(B)$), i.e.

$$m(C) = \begin{cases} m_e(C) & \text{if } C \leq B, \\ m_s(C) + (m_e(B) - m_s(B))e^{-\frac{C-B}{4B}} & \text{if } C > B. \end{cases}$$

Notice that, if we set $B = 0$, we obtain SHARDS$_{\text{adj}}$ scheme with constant sampling rate.

As for the parameters B and R_s, we provide a sensitivity analysis in Section 5, while we discuss the general guidelines for setting them in Section 6. Here we anticipate that in our experiments no particular tuning was required.

As for R_s, the considerations made by Waldspurger *et al.* [37] are valid also in our case. As for B, in our experiments we notice that, even for traces with millions of items, only few hundreds have very high popularity, and a value of B as low as 10^3 item references provides very accurate results. For the same trace, if we use $R_s = 0.01$, the memory necessary to hold the references to the sampled items is of the order of 10^4 item references, and $B = 10^3$ adds only a small fraction to that memory consumption.

The proposed scheme has a complexity $\mathcal{O}(\log R_s M)$, where M is the number of distinct items in the trace, which is due to accesses to $\mathcal{T}_s$. As for $\mathcal{T}_e$, since its size is constant (B item references), the cost for the data structure operations is $\mathcal{O}(1)$.

Constant Complexity Scheme. A fixed sampling rate has computational complexity and memory requirements that depend on the number of sampled items, which may grow as we consider longer and longer traces. Waldspurger *et al.* [37] propose an adaptive sampling method to maintain a $\mathcal{O}(1)$ complexity per requests. This can be achieved by fixing a priori the number $s_{\max}$ of items to sample, and then tracking the $s_{\max}$ items with smallest values of the hash function. As more requests are processed, the sampling rate implicitly converges to the minimum value required to maintain $s_{\max}$ references. Our mixed approach can be easily adapted in this direction, by fixing the size of $\mathcal{T}_s$.

5 Evaluation

5.1 Experimental Methodology

We compare our solution with the state-of-the-art approaches based on spatial sampling [37] [36], both when the sampling rate is fixed, and when the complexity is constant. If not otherwise stated, we consider the SHARDS$_{\text{adj}}$ variant

In case of fixed sampling rate, SHARDS$_{\text{adj}}$ adopts a sampling rate R. Given a trace with M distinct items, the scheme keeps track of RM items. For a fair comparison with our scheme, we adopt a sampling rate R_s that leads to keep track of the same number of item references, i.e., $B + R_s M = RM$. If not otherwise stated, we set $B = 1000$ and $R_s = R - B/M$. In most of our experiments, M is of the order of few millions, so with $R = 0.01$, R_s is slightly smaller than R. In Section 5.2 we will show the impact of B on the accuracy.

In case of constant complexity, SHARDS$_{\text{adj}}$ fixes the number of operations by maintaining a constant number of item references $s_{\max}$. This means that the scheme requires $2\log(s_{\max})$ operations per request (where the factor 2 is due to the additional data structure to track the $s_{\max}$ items with smallest hash). In our case, there is an additional cost of $\log B$ due to the exact portion of the MRC, so, to make a fair comparison, we set the size of $\mathcal{T}_s$ to a value of $s'_{\max}$ such that $\log B + 2\log(s'_{\max}) = 2\log(s_{\max})$, *i.e.*, $s'_{\max} = s_{\max}/\sqrt{B}$.

In the following sections, we will report simply R or $s_{\max}$ used for SHARDS$_{\text{adj}}$ [37]. The corresponding parameters of our scheme are computed as explained above.

5.2 IRM traces

We start testing our solution with the IRM traces described in Section 3. We focus on the two more problematic popularity

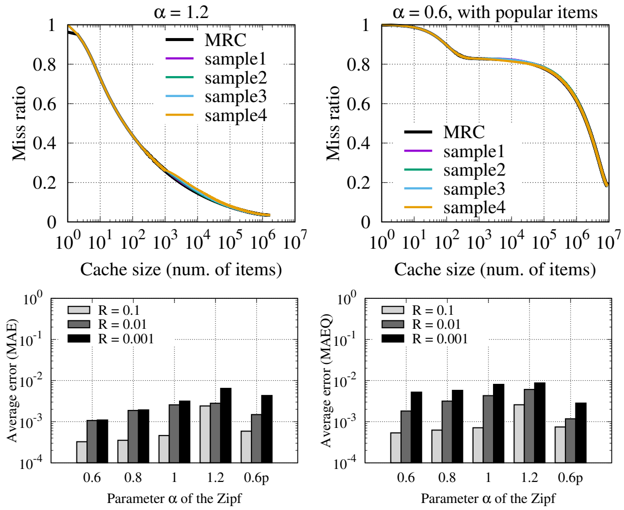


Figure 7: MRCs built from samples with our approach (top) and accuracy (MAE bottom left, MAEQ bottom right).

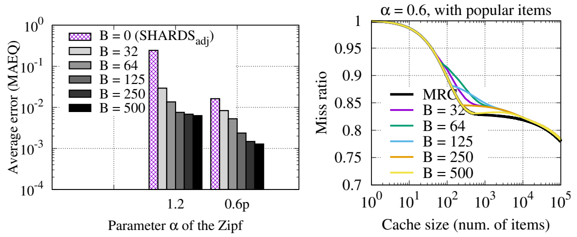


Figure 8: Accuracy for different values of B ($R = 0.01$), and the corresponding MRC ($\alpha = 0.6p$, zoom on the head of the MRC).

distributions: the Zipf one with $\alpha = 1.2$, and the Zipf one with $\alpha = 0.6$ modified by introducing 20 very popular items, labeled as "0.6p." Figure 7 shows the approximate MRC, along with the MAE and MAEQ. Our solution is able to build approximate MRCs using the same amount of memory as SHARDS$_{\text{adj}}$. The average error per quantile reaches at most 0.008 in case of an equivalent sampling rate as low as $R = 0.001$—the corresponding value of R_s is 0.0004 for $\alpha = 1.2$ and 0.0008 for the "0.6p" case. With SHARDS$_{\text{adj}}$, instead, the error with $R = 0.001$ was more than 40 times larger (0.35) for $\alpha = 1.2$ (Figure 3, right). For most of the settings, our solution also slightly improves the MAE, being equal only for $\alpha = 1.2$ and $R = 0.001$.

Impact of B. The proposed scheme has a parameter B, which is the maximum cache size considered for the exact MRC. Recall that, if $B = 0$, we obtain SHARDS$_{\text{adj}}$'s results. Figure 8, left, shows the impact of the accuracy for different values of B. We consider the default case with $R = 0.01$. Since the ratio B/M is less than 10^{-3}, then the equivalent R_s is approximately 0.01 too.

As we increase B, there is a significant error reduction. The reason can be seen looking at the approximate MRC, which is shown in Figure 8, right. Since we have a sampling rate $R_s \approx R = 0.01$, only one of the 100 most popular items, on average, is sampled, so the approximate MRC is very inaccurate in the range $[1, 100]$. As our algorithm uses the approximate sample-based MRC starting from $B + 1$, as long as B is smaller than 100 (*i.e.*, $1/R_s$), the error of the approximate MRC also affects the final MRC.

Table 1: Trace characteristics

name	year	# items	# req	reference
fiu	2008	6.1 M	14.3 M	[22]
ms-ex	2007	2.6 M	8.9 M	[18]
ms-dev	2007	6.3 M	18.2 M	[18]
systor	2016	12.7 M	34.3 M	[23]
CDN	2015	1.6 M	11.2 M	[8]

5.3 Real-world traces

We consider a set of publicly-available block I/O traces from SNIA IOTTA repository [31], along with traces from Akamai, a major CDN provider. Table 1 summarizes the characteristics of the traces. From the SNIA IOTTA repository, we have considered the most recent trace—labeled as *systor* [23]—along with older traces collected at FIU [22] and at Microsoft [18].

For experimental reproducibility, we report here the details of the traces. For the *fiu* traces [22], we consider the subtrace IODedup/Web. The *ms-ex* is the trace named "Microsoft Enterprise Traces, Exchange Server Traces" [18], which have been collected for Exchange server for a duration of 24-hours—we consider the first 3.5 hours. The *ms-dev* is the trace named "Microsoft Production Server Traces - Development Tools Release" [18]. The *systor* traces [23] collect requests for different block storage devices over 28 days: we consider one day (March, 9th) of the device called "LUN2." Finally, the *CDN* trace [8] contains multiple days of traffic, of which we consider portions of 6 hours—we tested different intervals finding similar qualitative results.

Request distribution. Figure 9 shows the empirical popularity distribution for two representative traces (*systor* and *CDN*), since the others are similar. The distributions show the presence of two distinct groups of items: a head with highly popular items, and a power-law tail—the figure shows the exponent α of the fitting power law distribution.

Approximate MRC with Constant Sampling Rate. Figure 10 shows the comparison between the exact MRC and the ones obtained with SHARDS$_{\text{adj}}$ (left column) and with our approach (right column) for some representative traces. In all cases we consider $R = 0.01$ and $B = 1000$. We have also computed the MAEQ (bottom subfigure).

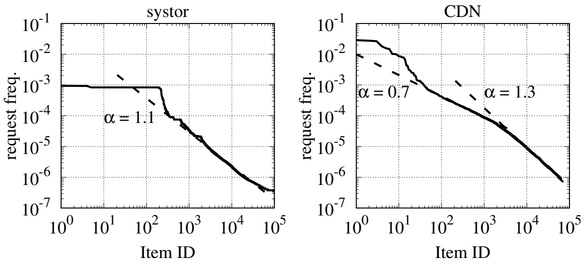


Figure 9: Item popularity distribution of the traces.

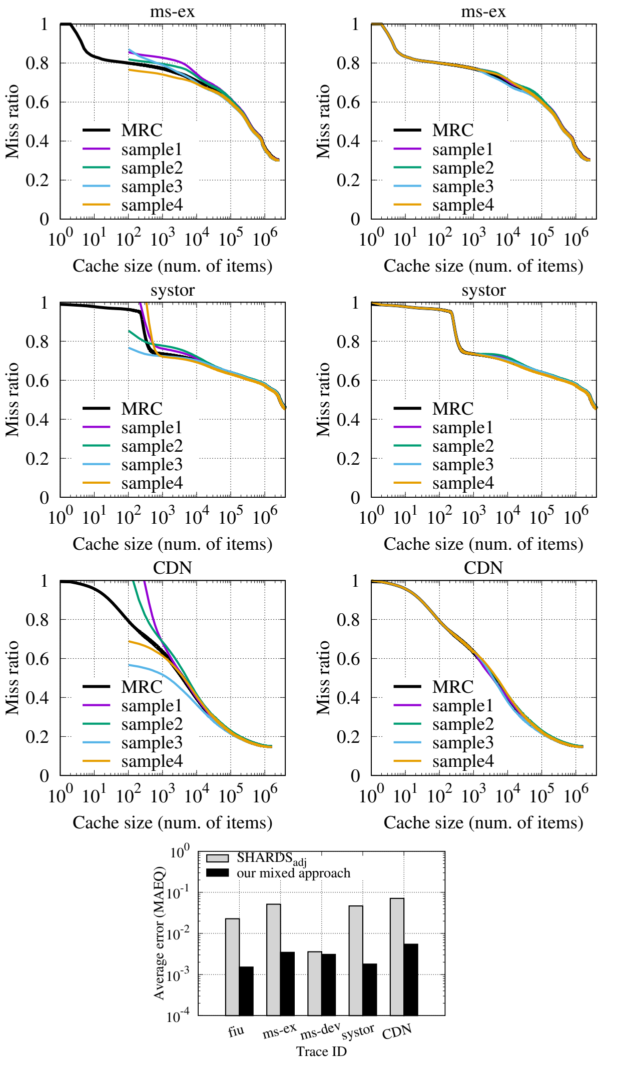


Figure 10: MRCs built from samples with $\text{SHARDS}_{\text{adj}}$ (left) and our approach (right). Bottom: MAEQ.

Except for the *ms-dev* trace, $\text{SHARDS}_{\text{adj}}$ fails to build an accurate MRC, with MAEQ in the range 0.05–0.10. With our approach, instead, the MAEQ is always below 0.01, and the accuracy can be appreciated visually comparing the approximate and exact MRCs. The *ms-dev* trace is representative of the case in which $\text{SHARDS}_{\text{adj}}$ provides good accuracy: our approach is able to provide equally accurate results.

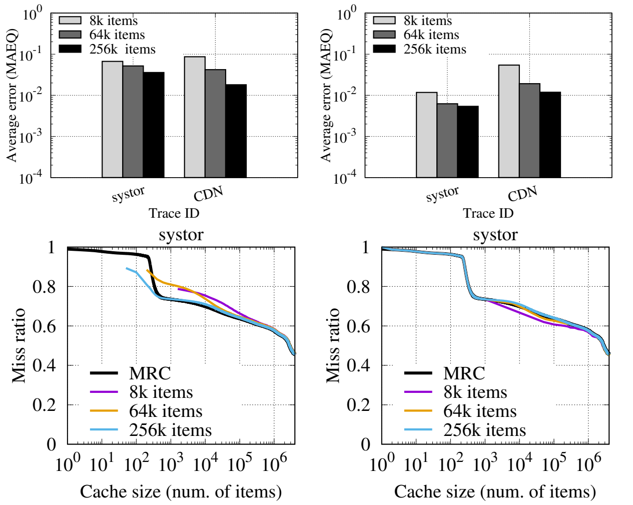


Figure 11: Top: Accuracy (MAEQ) for constant complexity with $\text{SHARDS}_{\text{adj}}$ (left) and our approach (right). Bottom: MRC of a representative trace, with $\text{SHARDS}_{\text{adj}}$ (left) and our approach (right).

Approximate MRC with Constant Complexity. We recall (Section 4) that a constant complexity per request is achieved by putting a cap on the number of sampled items s'_{max}, and then to the total memory used ($B + s'_{\text{max}}$). Figure 11 (top) shows the MAEQ with $\text{SHARDS}_{\text{adj}}$ and with our approach for different memory sizes—if not otherwise stated $B = 1000$. Figure 11 (bottom) shows the MRC. As the number of items increases, the head of the approximate MRC with $\text{SHARDS}_{\text{adj}}$ converges to the exact shape. With our approach, the MAEQ is smaller, because the first B positions are always correct.

Overheads. The experimental campaign has been specifically designed to compare our scheme and $\text{SHARDS}_{\text{adj}}$, and the memory used in both scenarios—constant sampling rate and constant complexity—as described in Section 5.1. We have evaluated the CPU usage using user and system time components as reported by `/usr/bin/time`. Our experiments confirm that $\text{SHARDS}_{\text{adj}}$ has a 75x speed up compared to the exact MRC computation [37]. Our scheme performs slightly worse with CPU usage on average 10%, and at most 20%, higher than $\text{SHARDS}_{\text{adj}}$. While the asymptotic computational complexity of the two schemes is the same, ours requires indeed a few more operations per request. Under $\text{SHARDS}_{\text{adj}}$, at each request, we need to compute the hash of the item identifier. With our scheme, in addition to this, we need to insert the item at the head of the tree data structure that keeps track of the first B positions (in case of a hit we first need to remove the item, but this does not happen at every request). The fact that CPU load only increases by 10% suggests that the overhead due to the additional insertion/deletion operations appears negligible in comparison to the hash computation cost. On a consumer laptop, our solution processed the traces in Table 1 (which span multiple hours) in less than 30s. It can the be used, not only offline on collected traces, but also online, as requests arrive. In the latter case, one may select a duration for the observation interval, e.g., one hour or less, and the MRC is computed at the end of the interval.

6 Discussion

Parameter configuration. In Section 5 we have presented a sensitivity analysis with respect to the parameter B, the sampling rate R_s (in case of constant sampling rate), and the maximum number of item references $B + s'_{\max}$ (in case of constant complexity). The choice of these parameters determines the amount of memory that will be used for the approximate MRC computation. Given the memory budget and using some simple characteristics of the trace, such as the number of items or the number of requests (which can also be estimated in an online setting), it is possible to estimate the maximum values for B and R_s.

We have already shown that, due to the specific way in which the final MRC is built, one should adopt a value $B \geq 1/R_s$ to avoid connecting the two MRCs at a point where the sampled one is very imprecise. Additional constraints, due to the specific context in which the MRCs are used, can drive the exact setting. For instance, if the cache needs to be split across different application types, their number and the total amount of storage available further limits B.

Extension to "non-stack" algorithms. The MRC construction technique in case of eviction policies that do not satisfy the inclusion property is different, *i.e.*, one needs to compute the miss ratios for different cache sizes in parallel, and then join the results. Waldspurger *et al.* [36] propose a general method where the miss ratio for cache size C is obtained simulating a cache with size RC with a request trace sampled with rate R. In their experiments they use the same scaling factor R for all the sizes, but our findings suggest that one wants to *differentiate* the sampling rate used, adopting a high (resp. low) sampling rate for small (resp. large) caches. The higher sampling rate for the small caches would be compensated by the smaller sampling rate used at large ones, so overall the memory requirement and the computational complexity could be maintained similar to the case with constant sampling rate.

Heterogeneous item size. Most of the work about MRC consider items with uniform sizes. In contrast, there are different scenarios, such as Web caches, where items have *heterogeneous sizes*. In this case, the MRC should inform the miss rate obtained for a given size of the cache in *bytes*, rather than in number of items. In order to build such a MRC, we need to modify the data structure used to keep track of the items in the cache as explained in Carra *et al.* [8].

Figure 12 (left) shows the exact and approximate MRC with SHARDS$_{\text{adj}}$ for the *CDN* trace. Notice that the X-axis now reports the cache size in MB. Figure 12 (right) shows the results with our approach. By combining the exact MRC with the one built from the samples, we are able to build a more accurate MRC using the same amount of memory as used by SHARDS$_{\text{adj}}$. The results are confirmed by the MAEQ, for which we obtain an average value of 0.007—almost one order of magnitude smaller than SHARDS$_{\text{adj}}$'s value (0.052).

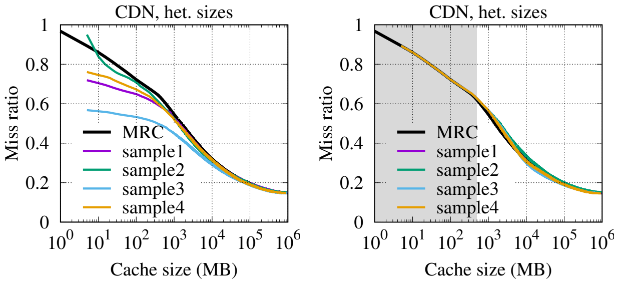


Figure 12: MRCs built from samples with SHARDS$_{\text{adj}}$ (left) and our approach (right—the shaded area corresponds to the portion of the exact MRC).

7 Conclusions

Sampling has been applied to calculate approximate MRCs with limited computational complexity. The use of such a technique requires a careful design in order to avoid the introduction of biases in the MRC construction. In this work, using a set of experiments and a model of a representative scenario, we studied the impact of popular items on the accuracy of the MRC, and we proposed a new approach that uses exact MRC calculation for small cache sizes while relying on sampling for large ones. The results using different real-world traces show that our solution is able to build approximate MRC with an error per quantile one order of magnitude smaller than state-of-the-art approaches, such as SHARDS$_{\text{adj}}$. As a future work, we plan to study how the parameters of our scheme should be set online depending on the characteristics of the request stream.

Acknowledgments

We would like to thank Pietro Michiardi for the initial discussions on this work, and for the computational resources kindly provided for part of the experiments. We thank the anonymous reviewers and our shepherd, Raju Rangaswami, for their constructive feedback. This work was partially supported by the Italian National Group for Scientific Computation (GNCS-INDAM) and by "Progetto di Eccellenza" of the Computer Science Dept., Univ. of Verona, Italy.

References

[1] Amazon Web Service ElastiCache. https://aws.amazon.com/elasticache/.

[2] Google Cloud Memorystore. https://cloud.google.com/memorystore/.

[3] Microsoft Azure Redis Cache. https://azure.microsoft.com/en-us/services/cache/.

[4] Erik Berg and Erik Hagersten. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004*, pages 20–27. IEEE, 2004.

[5] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[6] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *ACM SIGPLAN Notices*, volume 53, pages 84–95. ACM, 2018.

[7] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. TTL-based Cloud Caches. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 685–693. IEEE, 2019.

[8] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. Elastic provisioning of cloud caches: A cost-aware ttl approach. *IEEE/ACM Transactions on Networking*, 2020.

[9] Calin Cascaval and David A Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 150–159. ACM, 2003.

[10] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412. ACM, 2014.

[11] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[12] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[13] David Eklov and Erik Hagersten. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 55–65. IEEE, 2010.

[14] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[15] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

[16] Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(3):12, 2016.

[17] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, 2016.

[18] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production Windows servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128. IEEE, 2008.

[19] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 111–122. IEEE, 2004.

[20] WC King. Analysis of paging algorithms. In *Proc. IFIP 1971 Congress, Ljubljana*, pages 485–490. North-Holland, 1972.

[21] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60. IEEE, 2015.

[22] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[23] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of*

the 10th ACM International Systems and Storage Conference, SYSTOR '17, pages 13:1–13:11. ACM, 2017.

[24] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-cave: Effective ssd caching to improve virtual machine storage performance. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 103–112. IEEE, 2013.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[26] Giovanni Neglia, Damiano Carra, Mingdong Feng, Vaishnav Janardhan, Pietro Michiardi, and Dimitra Tsigkari. Access-time-aware cache algorithms. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 2(4):21, 2017.

[27] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[28] Sundaresan Rajasekaran, Shaohua Duan, Wei Zhang, and Timothy Wood. Multi-cache: Dynamic, efficient partitioning for multi-tier caches in consolidated vm environments. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 182–191. IEEE, 2016.

[29] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[30] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.

[31] SNIA. SNIA iotta repository block I/O traces. `http://iotta.snia.org/tracetypes/3`. Accessed: July 2019.

[32] Gokul Soundararajan, Jin Chen, Mohamed A Sharaf, and Cristiana Amza. Dynamic partitioning of the cache hierarchy in shared data centers. *Proceedings of the VLDB Endowment*, 1(1):635–646, 2008.

[33] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global CDN. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67. ACM, 2017.

[34] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. *ACM SIGARCH Computer Architecture News*, 37(1):121–132, 2009.

[35] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 283–294. ACM, 2011.

[36] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of USENIX ATC*, pages 487–498, 2017.

[37] Carl A Waldspurger, Nohhyun Park, Alexander T Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *FAST*, pages 95–110, 2015.

[38] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. Characterizing storage workloads with counter stacks. In *OSDI*, pages 335–349, 2014.

[39] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, pages 91–100. ACM, 2008.

[40] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31(6):1–39, 2009.

Can Applications Recover from `fsync` Failures?

Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin – Madison

Abstract

We analyze how file systems and modern data-intensive applications react to fsync failures. First, we characterize how three Linux file systems (ext4, XFS, Btrfs) behave in the presence of failures. We find commonalities across file systems (pages are always marked clean, certain block writes always lead to unavailability), as well as differences (page content and failure reporting is varied). Next, we study how five widely used applications (PostgreSQL, LMDB, LevelDB, SQLite, Redis) handle fsync failures. Our findings show that although applications use many failure-handling strategies, none are sufficient: fsync failures can cause catastrophic outcomes such as data loss and corruption. Our findings have strong implications for the design of file systems and applications that intend to provide strong durability guarantees.

1 Introduction

Applications that care about data must care about how data is written to stable storage. Issuing a series of write system calls is insufficient. A write call only transfers data from application memory into the operating system; the OS usually writes this data to disk lazily, improving performance via batching, scheduling, and other techniques [25,44,52,53].

To update persistent data correctly in the presence of failures, the order and timing of flushes to stable storage must be controlled by the application. Such control is usually made available to applications in the form of calls to fsync [9,47], which forces unwritten ("dirty") data to disk before returning control to the application. Most update protocols, such as write-ahead logging or copy-on-write, rely on forcing data to disk in particular orders for correctness [30,31,35,38,46,56].

Unfortunately, recent work has shown that the behavior of fsync during failure events is ill-defined [55] and error prone. Some systems, for example, mark the relevant pages clean upon fsync failure, even though the dirty pages have not yet been written properly to disk. Simple application responses, such as retrying the failed fsync, will not work as expected, leading to potential data corruption or loss.

In this paper, we ask and answer two questions related to this critical problem. The first question (§3) relates to the file system itself: why does fsync sometimes fail, and what is the effect on file-system state after the failure event?

To answer this first question, we run carefully-crafted micro-workloads on important and popular Linux file systems (ext4 [43], XFS [54], Btrfs [50]) and inject targeted block failures in the I/O stream. We then use a combination of tools to examine the results. Our findings show commonalities across file systems as well as differences. For example, all three file systems mark pages clean after fsync fails, rendering techniques such as application-level retry ineffective. However, the content in said clean pages varies depending on the file system; ext4 and XFS contain the latest copy in memory while Btrfs reverts to the previous consistent state. Failure reporting is varied across file systems; for example, ext4 data mode does not report an fsync failure immediately in some cases, instead (oddly) failing the subsequent call. Failed updates to some structures (e.g., journal blocks) during fsync reliably lead to file-system unavailability. And finally, other potentially useful behaviors are missing; for example, none of the file systems alert the user to run a file-system checker after the failure.

The second question we ask is (§4): how do important data-intensive applications react to fsync failures? To answer this question, we build CuttleFS, a FUSE file system that can emulate different file system fsync failures. CuttleFS maintains its own page cache in user-space memory, separate from the kernel page cache, allowing application developers to perform durability tests against characteristics of different file systems, without interference from the underlying file system and kernel.

With this test infrastructure, we examine the behavior of five widely-used data-management applications: Redis [18], LMDB [15], LevelDB [12], SQLite [20] (in both RollBack [1] and WAL modes [21]), and PostgreSQL [15] (in default and DirectIO modes). Our findings, once again, contain both specifics per system, as well as general results true across some or all. Some applications (Redis) are surprisingly careless with fsync, not even checking its return code before returning success to the application-level update; the result is a database with old, corrupt, or missing keys. Other applications (LMDB) exhibit false-failure reporting, returning an error to users even though on-disk state is correct. Many applications (Redis, LMDB, LevelDB, SQLite) exhibit data corruptions; for example, SQLite fails to write data to its rollback journal and corrupts in-memory state by reading from said journal when a transaction needs to be rolled back. While corruptions can cause some applications to reject newly inserted records (Redis, LevelDB, SQLite), both new and old data can be lost on updates (PostgreSQL). Finally, applications (LevelDB, SQLite, PostgreSQL) sometimes seemingly work correctly as long as the relevant data remains in the file-system cache; when said data is purged from the cache (due to cache pressure or OS restart), however, the application then returns stale data (as retrieved from disk).

We also draw high-level conclusions that take both file-system and application behavior into account. We find that applications expect file systems on an OS platform (e.g., Linux) to behave similarly, and yet file systems exhibit nuanced and important differences. We also find that applications employ numerous different techniques for handling fsync failures, and yet none are (as of today) sufficient; even after the PostgreSQL fsync problem was reported [55], no application yet handles its failure perfectly. We also determine that application recovery techniques often rely upon the file-system page cache, which does not reflect the persistent state of the system and can lead to data loss or corruption; applications should ensure recovery protocols only use existing persistent (on-disk) state to recover. Finally, in comparing ext4 and XFS (journaling file systems) with Btrfs (copy-on-write file system), we find that the copy-on-write strategy seems to be more robust against corruptions, reverting to older states when needed.

The rest of this paper is organized as follows. First, we motivate why this study is necessary (§2), followed by a file-system study (§3). Next, we study how applications react to fsync failures (§4). We then discuss the implications of our findings (§5), discuss related work (§6), and conclude (§7).

2 Motivation

Applications that manage data must ensure that they can handle and recover from any fault that occurs in the storage stack. Recently, a PostgreSQL user encountered data corruption after a storage error and PostgreSQL played a part in that corruption [17]. Because of the importance and complexity of this error, we describe the situation in detail.

PostgreSQL is an RDBMS that stores tables in separate files and uses a write-ahead log *(wal)* to ensure data integrity [16]. On a transaction commit, the entry is written to the log and the user is notified of the success. To ensure that the log does not grow too large (as it increases startup time to replay all entries in the log), PostgreSQL periodically runs a checkpoint operation to flush all changes from the log to the different files on disk. After an fsync is called on each of the files, and PostgreSQL is notified that everything was persisted successfully, the log is truncated.

Of course, operations on persistent storage do not always complete successfully. Storage devices can exhibit many different types of partial and transient failures, such as latent sector errors [27, 41, 51], corruptions [26], and misdirected writes [42]. These device faults are propagated through the file system to applications in a variety of ways [40, 49], often causing system calls such as read, write, and fsync to fail with a simple return code.

When PostgreSQL was notified that fsync failed, it retried the failed fsync. Unfortunately, the semantics for what should happen when a failed fsync is retried are not well defined. While POSIX aims to standardize behavior, it only states that outstanding IO operations are not guaranteed to have been completed in the event of failures during fsync [14]. As we shall see, on many Linux file systems, data pages that fail to be written, are simply marked clean in the page cache when fsync is called and fails. As a result, when PostgreSQL retried the fsync a second time, there were no dirty pages for the file system to write, resulting in the second fsync succeeding without actually writing data to disk. PostgreSQL assumed that the second fsync persisted data and continued to truncate the write-ahead log, thereby losing data. PostgreSQL had been using fsync incorrectly for 20 years [55].

After identifying this intricate problem, developers changed PostgreSQL to respond to the fsync error by crashing and restarting without retrying the fsync. Thus, on restart, PostgreSQL rebuilds state by reading from the *wal* and retrying the entire checkpoint process. The hope and intention is that this crash and restart approach will not lose data. Many other applications like WiredTiger/MongoDB [24] and MySQL [3] followed suit in fixing their fsync retry logic.

This experience leads us to ask a number of questions. As application developers are not certain about the underlying file-system state on fsync failure, the first part of our study answers what happens when fsync fails. How do file systems behave after they report that an fsync has failed? Do different Linux file systems behave in the same way? What can application developers assume about the state of their data after an fsync fails? Thus, we perform an in-depth study into the fsync operation for multiple file systems.

The second part of our study looks at how data-intensive applications react to fsync failures. Does the PostgreSQL solution indeed work under all circumstances and on all file systems? How do other data-intensive applications react to fsync failures? For example, do they retry a failed fsync, avoid relying on the page cache, crash and restart, or employ a different failure-handling technique? Overall, how well do applications handle fsync failures across diverse file systems?

3 File System Study

Our first study explores how file systems behave after reporting that an fsync call has failed. After giving a brief background of caching in file systems, we describe our methodology and our findings for the three Linux file systems.

3.1 Background

File systems provide applications with open, read, and write system calls to interact with the underlying storage media. Since block devices such as hard disks and solid state drives are much slower than main memory [57], the operating system maintains a page cache of frequently used pages of files in kernel space in main memory.

When an application calls read, the kernel first checks if the data is in the page cache. If not, the file system retrieves the data from the underlying storage device and stores it in the page cache. When an application calls write, the kernel only *dirties* the page in memory while notifying the application that the write succeeded; there is now a mismatch between the data in memory and on the device and data can potentially be lost. For durability, the file system periodically synchronizes content between memory and disk by *flushing* dirty pages and marking them clean. Applications that require stronger durability guarantees can force the dirty pages to disk using the fsync system call.

Applications can choose to bypass the page cache altogether by opening files with *O_DIRECT* (DirectIO). For caching, applications must perform their own in user space. Calls to fsync are still required since data may be cached within the underlying storage media; an fsync issues a FLUSH command to the underlying device so it pushes data all the way to stable storage.

3.2 Methodology

To understand how file systems should behave after reporting an fsync failure, we begin with the available documentation. The fsync man pages [9] report that fsync may fail for many reasons: the underlying storage medium has insufficient space (*ENOSPC* or *EDQUOT*), the file descriptor is not valid (*EBADF*), or the file descriptor is bound to a file that does not support synchronization (*EINVAL*). Since these errors can be discovered by validating input and metadata before initiating write operations, we do not investigate them further.

We focus on errors that are encountered only after the file system starts synchronizing dirty pages to disk; in this case, fsync signals an *EIO* error. *EIO* errors are difficult to handle because the file system may have already begun an operation (or changed state) that it may or may not be able to revert.

To trigger *EIO* errors, we consider single, transient, write faults in line with the fail-partial failure model [48,49]. When the file system sends a write request to the storage device, we inject a fault for a single sector or block within the request. Specifically, we build a kernel module device-mapper target that intercepts block-device requests from the file system and fails a particular write request to a particular sector or block while letting all other requests succeed; this allows us to observe the impact on an unmodified file system.

3.2.1 Workloads

To exercise the fsync path, we create two simple workloads that are representative of common write patterns seen in data-intensive applications.

Single Block Update (w_{su}): open an existing file containing three pages (12KB) and modify the middle page. This workload resembles many applications that modify the contents of existing files: LMDB always modifies the first two metadata pages of its database file; PostgreSQL stores tables as files on disk and modifies them in-place. Specifically, w_{su} issues system calls in the following sequence: open, lseek(4K), write(4K), *fsync*, *fsync*, sleep(40), close. The first fsync forces the dirty page to disk. While one fsync is sufficient in the absence of failures, we are interested in the impact of fsync retries after a failure; therefore, w_{su} includes a second fsync. Finally, since ext4, XFS, and Btrfs write out metadata and checkpoint the journal periodically, w_{su} includes a sleep for 40 seconds.

Multi Block Append (w_{ma}): open a file in append mode and write a page followed by an fsync; writing and fsyncing is repeated after sleeping. This workload resembles many applications that periodically write to a log file: Redis writes every operation that modifies its in-memory data structures to an append only file; LevelDB, PostgreSQL, and SQLite write to a write-ahead-log and fsync the file after the write. w_{ma} repeats these operations after a delay to allow checkpointing to occur; this is realistic as clients do not always write continuously and checkpointing may occur in those gaps. Specifically, w_{ma} issues system calls in the following sequence: open (in append mode), write(4K), *fsync*, sleep(40), write(4K), *fsync*, sleep(40), close.

3.2.2 Experiment Overview

We run the workloads on three different file systems: ext4, XFS, and Btrfs, with default mkfs and mount options. We evaluate both ext4 with metadata ordered journaling (data=ordered) and full data journaling (data=journal). We use an Ubuntu OS with Linux kernel version 5.2.11.

For each file system and workload, we first trace the block write access pattern. We then repeat the workload multiple times, each time configuring the fault injector to fail the i^{th} write access to a given sector or block. We only fail a single block or sector within the block in each iteration. We use a combination of offline tools (debugfs and xfs_db) and documentation to map each block to its respective file system data structure. We use SystemTap [22] to examine the state of relevant buffer heads and pages associated with data or metadata in the file system.

3.2.3 Behavior Inference

We answer the following questions for each file system:

Basics of fsync Failures:

Q1 Which block (data, metadata, journal) failures lead to fsync failures?

Q2 Is metadata persisted if a data block fails?

Q3 Does the file system retry failed block writes?

Q4 Are failed data blocks marked clean or dirty in memory?

Q5 Does in-memory page content match what is on disk?

Failure Reporting:

Q6 Which future fsync will report a write failure?

Q7 Is a write failure logged in the syslog?

After Effects of fsync Failure:

Q8 Which block failures lead to file-system unavailability?

Q9 How does unavailability manifest? Does the file system shutdown, crash, or remount in read-only mode?

Q10 Does the file suffer from holes or block overwrite failures? If so, in which parts of a file can they occur?[1]

Recovery:

Q11 If there is any inconsistency introduced due to fsync failure, can fsck detect and fix it?

3.3 Findings

We now describe our findings for the three file systems we have characterized: ext4, XFS, and Btrfs. Our answers to our posed questions are summarized in Table 1.

3.3.1 Ext4

The ext4 file system is a commonly-used journaling file system on Linux. The two most common options when mounting this file system are *data=ordered* and *data=journal* which enable ext4 ordered mode and ext4 data mode, respectively. Ext4 ordered mode writes metadata to the journal whereas ext4 data mode writes both data and metadata to the journal.

Ext4 ordered mode: We give an overview of ext4 ordered mode by describing how it behaves for our two representative workloads when no failures occur.

Single Block Update (w_{su}). When no fault is injected and fsync is successful, ext4 ordered mode behaves as follows. During the write (Step 1), ext4 updates the page in the page cache with the new contents and marks the page dirty. On fsync, the page is written to a data block; after the data-block write completes successfully, the metadata (i.e., the inode with a new modification time) is written to the journal, and fsync returns 0 indicating success (Step 2). After the fsync, the dirty page is marked clean and contains the newly written data. On the second fsync, as there are no dirty pages, no block writes occur, and as there are no errors, fsync returns 0 (Step 3). During sleep, the metadata in the journal is checkpointed to its final in-place block location (Step 4). No writes or changes in page state occur during the close (Step 5).

If fsync fails (i.e., returns -1 with errno set to *EIO*), a variety of write problems could have occurred. For example, the data-block write could have failed; if this happens, ext4 does not write the metadata to the journal. However, the updated page is still marked clean and contains the newly written data from Step 1, causing a discrepancy with the contents on disk. Furthermore, even though the inode table was not written to the journal at the time of the data fault, the inode table containing the updated modification time is written to the journal on the second fsync in Step 3. Steps 4 and 5 are the same as above, and thus the inode table is checkpointed.

Thus, applications that read this data block while the page remains in the page cache (i.e., the page has not been evicted and the OS has not been rebooted) will see the new contents of the data; however, when the page is no longer in memory and must be read from disk, applications will see the old contents.

Alternatively, if fsync failed, it could be because a write to one of the journal blocks failed. In this case, ext4 aborts the journal transaction and remounts the file system in read-only mode, causing all future writes to fail.

Multi Block Append (w_{ma}). This next workload exercises additional cases in the fsync error path. If there are no errors and all fsyncs are successful, the multi-block append workload on ext4 behaves as follows. First, during write, ext4 creates a new page with the new contents and marks it dirty (Step 1). On fsync, the page is written to a newly allocated on-disk data block; after the data-block write completes successfully, the relevant metadata (i.e., both the inode table and the block bitmap) are written to the journal, and fsync returns success (Step 2). As in w_{su}, the page is marked clean and contains the newly written data. During sleep, the metadata is checkpointed to disk (Step 3); specifically, the inode contains the new modification time and a link to the newly allocated block, and the block bitmap now indicates that the newly allocated block is in use. The pattern is repeated for the second write (Step 4), fsync (Step 5), and sleep (Step 6). As in w_{su}, there are no write requests or changes in page state during close (Step 7).

An fsync failure could again indicate numerous problems. First, a write to a data block could have failed in Step 2. If this is the case, the fsync fails and the page is marked clean; as in w_{su}, the page contains the newly written data, differing from the on-disk block that contains the original block contents. The inode table and block bitmap are written to disk in Step 3; thus, even though the data itself has not been written, the inode is modified to reference this block and the corresponding bit is set in the block bitmap. When the workload writes another 4KB of data in Step 4, this write continues oblivious of the previous fault and Steps 5, 6, and 7 proceed as usual.

Thus, with a data-block failure, the on-disk file contains a non-overwritten block where it was supposed to contain the data from Step 1. A similar possibility is that the write to a data block in Step 5 fails; in this case, the file has a non-overwritten block at the end instead of somewhere in the middle. Again, an application that reads any of these failed data blocks while they remain in the page cache will see the newly appended contents; however, when any of those pages are no longer in memory and must be read from disk, applications will read the original block contents.

An fsync failure could also indicate that a write to a journal-block failed. In this case, as in w_{su}, the fsync returns an error and the following write fails since ext4 has been remounted in read-only mode.

Because this workload contains an fsync after the metadata has been checkpointed in Step 3, it also illustrates the impact of faults when checkpointing the inode table and block bitmap. We find that despite the fact that a write has failed and

[1] In file-system terminology, a hole is a region in a file for which there is no block allocated. If a block is allocated but not overwritten with the new data, we consider the file to have a *non-overwritten block* and suffer from *block overwrite failure*.

		fsync Failure Basics					Error Reporting		After Effects			Recovery
		Which block failure causes fsync failure?	Is metadata persisted on data block failure?	Which block failures are retried?	Is the page dirty or clean after failure?	Does the in-memory content match disk?	Which fsync reports the failure?	Is the failure logged to syslog?	Which block failure causes unavailability?	What type of unavailability?	Holes or block overwrite failures? If yes where do they occur?	Can fsck help detect holes or block overwrite failures?
		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
ext4	ordered	data,jrnl	yes [A]		clean [B]	no [B]	immediate	yes	jrnl	remount-ro	NOB, anywhere [A]	no
	data	data,jrnl	yes [A]		clean [B]	no [B]	next [C]	yes	jrnl	remount-ro	NOB, anywhere [A]	no
XFS		data,jrnl	yes [A]	meta	clean [B]	no [B]	immediate	yes	jrnl,meta	shutdown	NOB, within [A]	no
Btrfs		data,jrnl	no		clean	yes	immediate	yes	jrnl,meta	remount-ro	HOLE, within [D]	yes

[A] Non-overwritten blocks (Q10) occur because metadata is persisted despite data-block failure (Q2).

[B] Marking a dirty page clean (Q4) even though the content does not match the disk (Q5) is problematic.

[C] Delayed reporting (Q6) of fsync failures may confuse application error-handling logic.

[D] Continuing to write to a file after an fsync failure is similar to writing to an offset greater than file size, causing a hole in the skipped portion (Q10).

Table 1: **Behavior of Various File Systems when fsync Fails.** *The table summarizes the behavior of the three file systems: ext4, XFS, and Btrfs according to the questions posed in Section 3.2.3. The questions are divided into four categories mentioned at the top. For questions that require identifying a block type, we use the following abbreviations: Data Block (data), Journal Block (jrnl), Metadata Block (meta). In Q9, Remount-**ro** denotes remounting in read-only mode. In Q10, "anywhere" and "within" describe the locations of the holes or non-overwritten blocks (NOB); "within" does not include the end of the file. Entries with a superscript denote a problem.*

the file system will now be in an inconsistent state, the following fsync does not return an error. However, the metadata error is logged to syslog.

We note that for none of these fsync failures does ext4 ordered mode recommend running the file system checker; furthermore, running the checker does not identify or repair any of the preceding problems. Finally, future calls to fsync never retry previous data writes that may have failed. These results for ext4 ordered mode are all summarized in Table 1.

The ext4 file system also offers functionality to abort the journal if an error occurs in a file data buffer (mount option data_err=abort) and remount the file system in read-only mode on an error (mount option errors=remount-ro). However, we observe that the results are identical with and without the mount options. [2]

Ext4 Data Mode: Ext4 data mode differs from ordered mode in that data blocks are first written to the journal and then later checkpointed to their final in-place block locations.

As shown in Table 1, the behavior of fsync in ext4 data mode is similar to that in ext4 ordered mode for most cases: for example, on a write error, pages may be marked clean even if they were not written out to disk, the file system is remounted in read-only mode on journal failures, meta-data failures are not reported by fsync, and files can end up with non-overwritten blocks in the middle or end.

However, the behavior of ext4 data mode differs in one important scenario. Because data blocks are first written to the journal and later to their actual block locations during checkpointing, the first fsync after a write may succeed even if a data block will not be successfully written to its permanent in-place location. As a result, a data-block fault causes the second fsync to fail instead of the first; in other words, the error reporting by fsync is delayed due to a *failed intention* [36].

3.3.2 XFS

XFS is a journaling file system that uses B-trees. Instead of performing physical journaling like ext4, XFS journals logical entries for changes in metadata.

As shown in Table 1, from the perspective of error reporting and fsync behavior, XFS is similar to that of ext4 ordered mode. Specifically, failing to write data blocks leads to fsync failure and the faulty data pages are marked clean even though they contain new data that has not been propagated to disk; as a result, applications that read this faulty data will see the new data only until the page has been evicted from the page cache. Similarly, failing to write a journal block will cause fsync failure, while failing to write a metadata block will not. XFS remains available for reads and writes after data-block faults.

XFS handles fsync failures in a few ways that are different than ext4 ordered mode. First, on a journal-block fault, XFS shuts down the file system entirely instead of merely remounting in read-only mode; thus, all subsequent read and write operations fail. Second, XFS retries metadata writes when it encounters a fault during checkpointing; the retry limit is determined by a value in /sys/fs/xfs/*/error/metadata/*/max_retries, but is infinite by default. If the retry limit is exceeded, XFS again shuts down the file system.

The multi-block append workload illustrates how XFS handles metadata when writes to related data blocks fail. If the write to the first data block fails, XFS writes no metadata to the journal and fails the fsync immediately. When later data blocks are successfully appended to this file, the metadata is updated which creates a non-overwritten block in the file corresponding to the first write. If instead, a write to a

[2] We verified our observations by reproducing them using standard Linux tools and have filed a bug report for the same [2].

data block contained in the last journal transaction fails, the on-disk metadata is not updated to reflect any of these last writes (i.e., the size of the file is not increased if any related blocks fail in the last transaction). [3] Thus, while in ext4 a failed write always causes a non-overwritten block, in XFS, non-overwritten blocks cannot exist at the end of a file. However, for either file system, if the failed blocks remain in the page cache, applications can read those blocks regardless of whether they are in the middle or the end of a file.

3.3.3 Btrfs

Btrfs is a copy-on-write file system that avoids writing to the same block twice except for the superblock which contains root-node information. At a high level, some of the actions in Btrfs are similar to those in a journaling file system: instead of writing to a journal, Btrfs writes to a log tree to record changes when an `fsync` is performed; instead of checkpointing to fixed in-place locations, Btrfs writes to new locations and updates the roots in its superblock. However, since Btrfs is based on copy-on-write, it has a number of interesting differences in how it handles `fsync` failures compared to ext4 and XFS, as shown in Table 1.

Like ext4 ordered mode and XFS, Btrfs fails `fsync` when it encounters data-block faults. However, unlike ext4 and XFS, Btrfs effectively reverts the contents of the data block (and any related metadata) back to its old state (and marks the page clean). Thus, if an application reads the data after this failure, it will never see the failed operation as a temporary state. As in the other file systems, Btrfs remains available after this data-block fault.

Similar to faults to the journal in the other file systems, in Btrfs, faults to the log-tree result in a failed `fsync` and a remount in read-only mode. Unlike ext4 and XFS, faults in the metadata blocks during checkpointing result in a remount in read-only mode (but `fsync` still does not return an error).

The multi-block append workload illustrates interesting behavior in Btrfs block allocation. If the first append fails, the state of the file system, including the B-tree that tracks all free blocks, is reverted. However, the next append will continue to write at the (incorrectly) updated offset stored in the file descriptor, creating a hole in the file. Since the state of the B-tree was reverted, the deterministic block allocator will choose to allocate the same block again for the next append operation. Thus, if the fault to that particular block was transient, the next `write` and `fsync` will succeed and there will simply be a one block hole in the file. If the fault to that particular block occurs multiple times, future writes will continue to fail; as a result, Btrfs may cause more holes within a file than ext4 and XFS. However, unlike ext4 and XFS, the file does not have block overwrite failures.

3.3.4 File System Summary

We now present a set of observations for the file systems based on the questions from Section §3.2.3.

File System Behavior to `fsync` Failures. On all the three file systems, only data and journal-block failures lead to `fsync` failures (Q1). Metadata-block failures do not result in `fsync` failures as metadata blocks are written to the journal during an `fsync`. However, during a checkpoint, any metadata failure on XFS and Btrfs lead to unavailability (Q8) while ext4 logs the error and continues.[4]

On both modes of ext4 and XFS, metadata is persisted even after the file system encounters a data-block failure (Q2); timestamps are always updated in both the file systems. Additionally, ext4 appends a new block to the file and updates the file size while XFS does so only when followed by a future successful `fsync`. As a result, we find non-overwritten blocks in both the middle and end of files for ext4, but in only the middle for XFS (Q10). Btrfs does not persist metadata after a data-block failure. However, because the process file-descriptor offset is incremented, future `writes` and `fsyncs` cause a hole in the middle of the file (Q10).

Among the three, XFS is the only file system that retries metadata-block writes. However, none of them retry data or journal-block writes (Q3).

All the file systems mark the page clean even after `fsync` fails (Q4). In both modes of ext4 and XFS, the page contains the latest write while Btrfs reverts the in-memory state to be consistent with what is on disk (Q5).

We note that even though all the file systems mark the page clean, this is not due to any behavior inherited from the VFS layer. Each file system registers its own handlers to write pages to disk (`ext4_writepages`, `xfs_vm_writepages`, and `btrfs_writepages`). However, each of these handlers call `clear_page_dirty_for_io` before submitting the bio request and do not set the dirty bit in case of failure in order to avoid memory leaks[5], replicating the problem independently.

Failure Reporting. While all file systems report data-block failures by failing `fsync`, ext4 ordered mode, XFS, and Btrfs fail the immediate `fsync`. As ext4 data mode puts data in the journal, the first `fsync` succeeds and the next `fsync` fails. (Q6). All block write failures, irrespective of block type are logged in the syslog (Q7).

After Effects. Journal block failures always lead to file-system unavailability. On XFS and Btrfs, metadata-block failures do so as well (Q8). While ext4 and Btrfs remount in read-only mode, XFS shuts down the file system (Q9). Holes and non-overwritten blocks (Q10) have been covered previously as part of Q2.

Recovery. None of the file systems alert the user to run a

[3] To be precise, the mtime and ctime of the file are updated, but not the size of the file. Additional experiments removed for space confirm this behavior.

[4] Ext4's error handling behavior for metadata has unintended side-effects but we omit the results as the rest of the paper focuses on data-block failures.

[5] Ext4 focuses on the common case of users removing USB sticks while still in use. Dirty pages that can never be written to the removed USB stick have to be marked clean to unmount the file system and reclaim memory [23].

file-system checker. However, the Btrfs checker is capable of detecting holes in files (Q11).

4 Application Study

We now focus on how applications are affected by `fsync` failures. In this section, we first describe our fault model with CuttleFS, followed by a description of the workloads, execution environment, and the errors we look for. Then, we present our findings for five widely used applications: Redis (v5.0.7), LMDB (v0.9.24), LevelDB (v1.22), SQLite (v3.30.1), and PostgreSQL (v12.0).

4.1 CuttleFS

We limit our study to how applications are affected by data-block failures as journal-block failures lead to unavailability and metadata-block failures do not result in `fsync` failures (§3.3). Our fault model is simple: when an application writes data, we inject a single fault to a data block or a sector within it.

We build CuttleFS[6] - a FUSE [39] file system to emulate the different file-system reactions to failures defined by our fault model. Instead of using the kernel's page cache, CuttleFS maintains its own page cache in user-space memory. Write operations modify user-space pages and mark them dirty while read operations serve data from these pages. When an application issues an `fsync` system call, CuttleFS synchronizes data with the underlying file system.

CuttleFS has two modes of operation: trace mode and fault mode. In trace mode, CuttleFS tracks writes and identifies which blocks are eventually written to disk. This is different from just tracing a `write` system call as an application may write to a specific portion of a file multiple times before it is actually flushed to disk.

In fail mode, CuttleFS can be configured to fail the ith write to a sector or block associated with a particular file. On `fsync` failure, as CuttleFS uses in-memory buffers, it can be directed to mark a page clean or dirty, keep the latest content, or revert the file to the previous state. Error reporting behavior can be configured to report failures immediately or on the next `fsync` call. In short, CuttleFS can react to `fsync` failures in any of the ways mentioned in Table 1 (Q4,5,6). Additionally, CuttleFS accepts commands to evict all or specific clean pages.

We configure CuttleFS to emulate the failure reactions of the file systems studied in Section 3.3. For example, in order to emulate ext4 ordered mode and XFS (as they both have similar failure reactions), we configure CuttleFS to mark the page clean, keep the latest content, and report the error immediately. Henceforth, when presenting our findings and referring to characteristics emulated by CuttleFS, we use CuttleFS$_{\text{ext4o,xfs}}$ for the above configuration. When the page is marked clean, has the latest content, but the error is reported on the next `fsync`, we use CuttleFS$_{\text{ext4d}}$. When the page is marked clean, the content matches what is on disk, and the error is reported immediately, we refer to it as CuttleFS$_{\text{btrfs}}$.

4.2 Workloads and Execution Environment

We run CuttleFS in trace mode and identify which blocks are written to by an application. For each application, we choose a simple workload that inserts a single key-value pair, a commonly used operation in many applications. We perform experiments both with an existing key (update) as well as a new key (insert). The keys can be of size 2B or 1KB.[7] The values can be of size 2B or 12KB. We run experiments for all four combinations. The large keys allow for the possibility of failing a single sector within the key and large values for pages within a value. Since SQLite and PostgreSQL are relational database management systems, we create a single table with two columns: keys and values.

Using the trace, we generate multiple failure sequences for each of the identified blocks and sectors within them. We then repeat the experiment multiple times with CuttleFS in fault mode, each time with a different failure sequence and file-system reaction. In order to observe the effects after a fault, we dump all key-value pairs before and after the workload.

We look for the following types of errors when performing the experiments:

- **OldValue (OV):** The system returns the new value for a while but then reverts to an old value, or the system conveys a successful response but returns the old value later on.
- **FalseFailure (FF):** The system informs the user that the operation failed but returns the new value in the future.
- **KeyCorruptions (KC)** and **ValueCorruptions (VC):** Corrupted keys or values are obliviously returned.
- **KeyNotFound (KNF):** The system informs the user that it has successfully inserted a key but it cannot be found later on, or the system fails to update a key to a new value but the old key-value pair disappears as well.

We also identify the factors within the execution environment that cause all these errors to be manifested. If an application maintains its own in-memory data structures, some errors may occur only when an application restarts and rebuilds in-memory state from the file system. Alternatively, the manifestation of these errors may depend on state changes external to the application, such as a single page eviction or a full page cache flush. We encode these different scenarios as:

- **App=KeepGoing:** The application continues without restarting.
- **App=Restart:** The application restarts either after a crash or a graceful shutdown. This forces the application to rebuild in-memory state from disk.

[6] Cuttlefish are sometimes referred to as the "chameleons of the sea" because of their ability to rapidly alter their skin color within a second. CuttleFS can change characteristics much faster.

[7] As LMDB limits key sizes to 511B, we use key sizes of 2B and 511B for LMDB experiments.

	A=KeepGoing	A=Restart
BC=Keep	\	/
BC=Evict	∣	−

Applications		ext4o,xfs = { clean differs immediate					ext4d = { clean differs next `fsync`					btrfs = { clean matches immediate				
		OV	**FF**	**KC**	**VC**	**KNF**	**OV**	**FF**	**KC**	**VC**	**KNF**	**OV**	**FF**	**KC**	**VC**	**KNF**
Redis		−		−	−	−	−		−	−	−	⌿				⌿
LMDB			−				+			+						
LevelDB			/				−		+	+	+					
SQLite	Rollback		+		+			+		−	−		✳			
	WAL		/							−	−					
PostgreSQL	Default		⌿				−				−					
	Direct I/O						−				−					

Table 2: **Findings for Applications on `fsync` Failure.** *The table lists the different types of errors that manifest for applications when `fsync` fails due to a data-block write fault. The errors (OV, FF, KC, VC, KNF) are described in §4.2. We group columns depending on how a file system reacts to an `fsync` failure according to our findings in §3.3 for Q4, Q5, and Q6. For example, both ext4 ordered and XFS (ext4o,xfs) mark a page **clean**, the page **differs** in in-memory and on-disk content, and the `fsync` failure is reported **immediately**. For each application, we describe when the error manifests, in terms of combinations of the four different execution environment factors (§4.2) whose symbols are provided at the top left corner. For example, OldValue manifests in Redis in the first group (ext4-ordered, XFS) only on (A)App=Restart,(BC)BufferCache=Evict. However, in the last group (Btrfs), the error manifests both on App=Restart,BufferCache=Evict as well as App=Restart,BufferCache=Keep, depicted as a combination of the two symbols.*

- **BufferCache=Keep:** No evictions take place.
- **BufferCache=Evict:** One or more clean pages are evicted.

Note that BufferCache=Evict can manifest by clearing the entire page cache, restarting the file system, or just evicting clean pages due to memory pressure. A full system restart would be the combination of App=Restart and BufferCache=Evict, which causes a loss of both clean and dirty pages in memory while also forcing the application to restart and rebuild state from disk.

Configuring CuttleFS to fail a certain block and react according to one of the file-system reactions while the application runs only addresses App=KeepGoing and BufferCache=Keep. The remaining three scenarios are addressed as follows. To simulate App=Restart and BufferCache=Keep, we restart the application and dump all key-value pairs, ensuring that no page in CuttleFS is evicted. To address the remaining two scenarios, we instruct CuttleFS to evict clean pages for both App=KeepGoing and App=Restart.

4.3 Findings

We configured all five applications to run in the form that offers most durability and discuss what they are in their respective sections. Table 2 summarizes the per-application results across different failure characteristics.

Note that these results are only for the simple workload that inserts a single key-value pair. A complex workload may exhibit more errors or mask the ones we observe.

Redis: Redis is an in-memory data-structure store, used as a database, cache, and message broker. By default, it periodically snapshots in-memory state to disk. However, for better durability guarantees, it provides options for writing every operation that modifies the store to an append-only file *(aof)* [19] and how often to `fsync` the *aof*. In the event of a crash or restart, Redis rebuilds in-memory state by reading the contents of the *aof*.

We configure Redis to `fsync` the file for every operation, providing strong durability. Thus, whenever Redis receives a request like an insert operation that modifies state, it writes the request to the *aof* and calls `fsync`. However, Redis trusts the file system to successfully persist the data and does not check the `fsync` return code. Regardless of whether `fsync` fails or not, Redis returns a successful response to the client.

As Redis returns a successful response to the client irrespective of `fsync` failure, FalseFailures do not occur. Since Redis reads from disk only when rebuilding in-memory state, errors may occur only during App=Restart.

On CuttleFS$_{\text{ext4o,xfs}}$ and CuttleFS$_{\text{ext4d}}$, Redis exhibits OldValue, KeyCorruption, ValueCorruption, and KeyNotFound errors. However, as seen in Table 2, these errors occur only on BufferCache=Evict and App=Restart. On BufferCache=Keep, the page contains the latest write which allows Redis to rebuild the latest state. However, when the page is evicted, future reads will force a read from disk, causing Redis to read whatever is on that block. OldValue and KeyNotFound errors manifest when a fault corrupts the *aof* format. When Redis restarts, it either ignores these entries when scanning the *aof*, or recommends running the *aof checker* which truncates the file to the last non-corrupted entry. A KeyCorruption and ValueCorruption manifest when the fault is within the key or value portion of the entry.

On CuttleFS$_{\text{btrfs}}$, Redis exhibits OldValue and KeyNotFound errors. These errors occur on App=Restart, regardless of buffer-cache state. When Redis restarts, the entries are missing from the *aof* as the file was reverted, and thus, the insert or update operation is not applied.

LMDB: Lightning Memory-Mapped Database (LMDB) is an embedded key-value store which uses B+Tree data structures whose nodes reside in a single file. The first two pages of the file are metadata pages, each of which contain a transaction ID and the location of the root node. Readers always use the metadata page with the latest transaction ID while writers make changes and update the older metadata page.

LMDB uses a copy-on-write bottom-up strategy [13] for committing write transactions. All new nodes from leaf to root are written to unused or new pages in the file, followed by an `fsync`. An `fsync` failure terminates the operation without updating the metadata page and notifies the user. If `fsync` succeeds, LMDB proceeds to update the old metadata page with the new root location and transaction ID, followed by another `fsync`.[8] If `fsync` fails, LMDB writes an old transaction ID to the metadata page in memory, preventing future readers from reading it.

On CuttleFS$_{\text{ext4o,xfs}}$, LMDB exhibits FalseFailures. When LMDB writes the metadata page, it only cares about the transaction ID and new root location, both of which are contained in a single sector. Thus, even though the sector is persisted to disk, failures in the seven other sectors of the metadata page can cause an `fsync` failure. As mentioned earlier, LMDB writes an old transaction ID (say ID1) to the metadata page in memory and reports a failure to the user. However, on BufferCache=Evict and App=Restart (such as a machine crash and restart), ID1 is lost as it was only written to memory and not persisted. Thus, readers read from the latest transaction ID which is the previously failed transaction.

LMDB does not exhibit FalseFailures in CuttleFS$_{\text{ext4d}}$ as the immediate successful `fsync` results in a success to the client. Instead, ValueCorruptions and OldValue errors occur on BufferCache=Evict, regardless of whether the application restarts or not. ValueCorruptions occur when a block containing a part of the value experiences a fault. As LMDB *mmaps*() the file and reads directly from the page cache, BufferCache=Evict such as a page eviction leads to reading the value of the faulted block from disk. OldVersion errors occur when the metadata page experiences a fault. The file system responds with a successful `fsync` initially (as data is successfully stored in the ext4 journal). For a short time, the metadata page has the latest transaction ID. However, when the page is evicted, the metadata page reverts to the old transaction ID on disk, resulting in readers reading the old value. KeyCorruptions do not occur as the maximum allowed key size is 511B.

As CuttleFS$_{\text{btrfs}}$ reports errors immediately, it does not face the problems seen in CuttleFS$_{\text{ext4d}}$. FalseFailures do not occur as the file is reverted to its previous consistent state. We observe this same pattern in many of the applications and omit them from the rest of the discussion unless relevant.

LevelDB: LevelDB is a widely used key-value store based on LSM trees. It stores data internally using MemTables and SSTables [33]. Additionally, LevelDB writes operations to a log file before updating the MemTable. When a MemTable reaches a certain size, it becomes immutable and is written to a new file as an SSTable. SSTables are always created and never modified in place. On a restart, if a log file exists, LevelDB creates an SSTable from its contents.

We configure LevelDB to `fsync` the log after every write, for stronger durability guarantees. If `fsync` fails, the MemTable is not updated and the user is notified about the failure. If `fsync` fails during SSTable creation, the operation is cancelled and the SSTable is left unused.

On CuttleFS$_{\text{ext4o,xfs}}$, as seen in Table 2, LevelDB exhibits FalseFailures only on App=Restart with BufferCache=Keep. When LevelDB is notified of `fsync` failure to the log file, the user is notified of the failure. However, on restart, since the log entry is in the page cache, LevelDB includes it while creating an SSTable from the log file. Read operations from this point forward return the new value, reflecting FalseFailures. FalseFailures do not occur on BufferCache=Evict as LevelDB is able to detect invalid entries through CRC checksums [33]. Faults in the SSTable are detected immediately and do not cause any errors as the newly generated SSTable is not used by LevelDB in case of a failure.

On CuttleFS$_{\text{ext4d}}$, LevelDB exhibits KeyNotFound and OldVersion errors when faults occur in the log file. When inserting a key-value pair, `fsync` returns successfully, allowing future read operations to return the new value. However, on BufferCache=Evict and App=Restart, LevelDB rejects the corrupted log entry and returns the old value for future read operations. Depending on whether we insert a new or existing key, we observe KeyNotFound or OldVersion errors when the log entry is rejected. Additionally, LevelDB exhibits KeyCorruption, ValueCorruption, and KeyNotFound errors for faults that occur in the SSTables. Ext4 data mode may only place the data in the journal and return a successful `fsync`. Later, during checkpointing, the SSTable is corrupted due to the fault. These errors manifest only on BufferCache=Evict, either while the application is running or on restart, depending on when the SSTable is read from disk.

SQLite: SQLite is an embedded RDBMS that uses BTree data structures. A separate BTree is used for each table and index but all BTrees are stored in a single file on disk, called the "main database file" *(maindb)*. During a transaction, SQLite stores additional information in a second file called the "rollback journal" *(rj)* or the "write-ahead log" *(wal)* depending on which mode it is operating in. In the event of a crash or restart, SQLite uses these files to ensure that committed or rolled-back transactions are reflected in the *maindb*. Once a transaction completes, these files are deleted. We perform experiments for both modes.

SQLite RollBack: In rollback journal mode, before SQLite modifies its user-space buffers, it writes the original contents to the *rj*. On commit, the *rj* is `fsyncd`. If it succeeds, SQLite writes a header to the *rj* and `fsyncs` again (2 `fsyncs` on the *rj*). If a fault occurs at this point, only the state in the user-space buffers need to be reverted. If not, SQLite proceeds to write to the *maindb* so that it reflects the state of the user-space buffers. *maindb* is then `fsyncd`. If the `fsync`

[8] To be precise, LMDB does not do a write followed by an `fsync` for metadata page updates. Instead, it uses a file descriptor that is opened in O_SYNC mode. On a write, only the metadata page is flushed to disk. On failure, it uses a normal file descriptor.

fails, SQLite needs to rewrite the old contents to the *maindb* from the *rj* and revert the state in its user-space buffers. After reverting the contents, the *rj* is deleted.

On CuttleFS$_{\text{ext4o,xfs}}$, SQLite Rollback exhibits FalseFailures and ValueCorruptions on BufferCache=Evict, regardless of whether the application restarts or not. When faults occur in the *rj*, SQLite chooses to revert in-memory state using the *rj* itself as it contains just enough information for a rollback of the user-space buffers. This approach works well as long as the latest contents are in the page cache. However, on BufferCache=Evict, when SQLite reads the *rj* to rollback in-memory state, the *rj* does not contain the latest write. As a result, SQLite's user-space buffers can still have the new contents (FalseFailure) or a corrupted value, depending on where the fault occurs.

SQLite Rollback exhibits FalseFailures in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above as the `fsync` failure is caught on the second `fsync` to the *rj*. Additionally, due to the late error reporting in CuttleFS$_{\text{ext4d}}$, SQLite Rollback exhibits ValueCorruption and KeyNotFound errors when faults occur in the *maindb*. SQLite sees a successful `fsync` after writing data to the *maindb* and proceeds to delete the *rj*. However, on App=Restart and BufferCache=Evict, the above mentioned errors manifest depending on where the fault occurs.

On CuttleFS$_{\text{btrfs}}$, SQLite Rollback exhibits FalseFailures for the same reasons mentioned above. However, they occur irrespective of whether buffer-cache state changes due to the fact that the contents in the *rj* are reverted. As there is no data in the *rj* to recover from, SQLite leaves the user-space buffers untouched. ValueCorruptions cannot occur as no attempt is made to revert the in-memory content.

SQLite WAL: Unlike SQLite Rollback, changes are written to a write-ahead log *(wal)* on a transaction commit. SQLite calls `fsync` on the *wal* and proceeds to change in-memory state. If `fsync` fails, SQLite immediately returns a failure to the user. If SQLite has to restart, it rebuilds state from the *maindb* first and then changes state according to the entries in the *wal*. To ensure that the *wal* does not grow too large, SQLite periodically runs a Checkpoint Operation to modify *maindb* with the contents from the *wal*.

On CuttleFS$_{\text{ext4o,xfs}}$, as seen in Table 2, SQLite WAL exhibits FalseFailures only on App=Restart with BufferCache=Keep, for reasons similar to LevelDB. It reads valid log entries from the page cache even though they might be invalid due to faults on disk.

On CuttleFS$_{\text{ext4d}}$, SQLite WAL exhibits ValueCorruption and KeyNotFound Errors when there are faults in the *maindb* during a Checkpoint Operation for the same reasons mentioned in SQLite Rollback.

PostgreSQL: PostgreSQL is an object-relational database system that maintains one file per database table. On startup, it reads the on-disk tables and populates user-space buffers. Similar to SQLite WAL, PostgreSQL reads entries from the write-ahead log *(wal)* and modifies user-space buffers accordingly. Similar to SQLite WAL, PostgreSQL runs a checkpoint operation, ensuring that the *wal* does not grow too large. We evaluate two configurations of PostgreSQL: the default configuration and a DirectIO configuration.

PostgreSQL Default: In the default mode, PostgreSQL treats the *wal* like any other file, using the page cache for reads and writes. PostgreSQL notifies the user of a successful *commit* operation only after an `fsync` on the *wal* succeeds. During a checkpoint, PostgreSQL writes data from its user-space buffers into the table and calls `fsync`. If the `fsync` fails, PostgreSQL, aware of the problems with `fsync` [8], chooses to crash. Doing so avoids truncating the *wal* and ensures that checkpointing can be retried later.

On CuttleFS$_{\text{ext4o,xfs}}$, PostgreSQL exhibits FalseFailures for reasons similar to LevelDB. While App=Restart is necessary to read the entry from the log, BufferCache=Evict is not. Further, the application restart cannot be avoided as PostgreSQL intentionally crashes on an `fsync` failure. On BufferCache=Keep, PostgreSQL reads a valid log entry in the page cache. On BufferCache=Evict, depending on which block experiences the fault, PostgreSQL either accepts or rejects the log entry. FalseFailures manifest when PostgreSQL accepts the log entry. However, if the file system were to also crash and restart, the page cache would match the on-disk state, causing PostgreSQL to reject the log entry. Unfortunately, ext4 currently does not behave as expected with mount options `data_err=abort` and `errors=remount-ro` (§3.3.1).

Due to the late error reporting in CuttleFS$_{\text{ext4d}}$, as seen in Table 2, PostgreSQL exhibits OldVersion and KeyNotFound Errors when faults occur in the database table files. As PostgreSQL maintains user-space buffers, these errors manifest only on BufferCache=Evict with App=Restart. During a checkpoint operation, PostgreSQL writes the user-space buffers to the table. As the fault is not yet reported, the operation succeeds and the *wal* is truncated. If the page corresponding to the fault is evicted and PostgreSQL restarts, it will rebuild its user-space buffers using an incorrect on-disk table file. The errors are exhibited depending on where the fault occurs. While KeyNotFound errors occur in other applications when a new key is inserted, PostgreSQL *loses existing keys on updates* as it modifies the table file in-place.

PostgreSQL DIO: In the DirectIO mode, PostgreSQL bypasses the page cache and writes to the *wal* using DirectIO. The sequence of operations during a transaction commit and a checkpoint are exactly the same as the default mode.

FalseFailures do not occur as the page cache is bypassed. However, OldVersion and KeyNotFound errors still occur in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above as writes to the database table files do not use DirectIO.

5 Discussion

We now present a set of observations and lessons for handling `fsync` failures across file systems and applications.

#1: Existing file systems do not handle fsync failures uniformly. In an effort to hide cross-platform differences, POSIX is intentionally vague on how failures are handled. Thus, different file systems behave differently after an `fsync` failure (as seen in Table 1), leading to non-deterministic outcomes for applications that treat all file systems equally. *We believe that the POSIX specification for `fsync` needs to be clarified and the expected failure behavior described in more detail.*

#2: Copy-on-Write file systems such as Btrfs handle fsync failures better than existing journaling file systems like ext4 and XFS. Btrfs uses new or unused blocks when writing data to disk; the entire file system moves from one state to another on success and no in-between states are permitted. Such a strategy defends against corruptions when only some blocks contain newly written data. *File systems that use copy-on-write may be more generally robust to `fsync` failures than journaling file systems.*

#3: Ext4 data mode provides a false sense of durability. Application developers sometimes choose to use a data journaling file system despite its lower performance because they believe data mode is more durable [11]. Ext4 data mode does ensure data and metadata are in a "consistent state", but only from the perspective of the file system. As seen in Table 2, application-level inconsistencies are still possible. Furthermore, applications cannot determine whether an error received from `fsync` pertains to the most recent operation or an operation sometime in the past. *When failed intentions are a possibility, applications need a stronger contract with the file system, notifying them of relevant context such as data in the journal and which blocks were not successfully written.*

#4: Existing file-system fault-injection tests are devoid of workloads that continue to run post failure. While all file systems perform fault-injection tests, they are mainly to ensure that the file system is consistent after encountering a failure. Such tests involve shutting down the file system soon after a fault and checking if the file system recovers correctly when restarted. *We believe that file-system developers should also test workloads that continue to run post failure, and see if the effects are as intended.* Such effects should then be documented. File-system developers can also quickly test the effect on certain characteristics by running those workloads on CuttleFS before changing the actual file system.

#5: Application developers write OS-specific code, but are not aware of all OS-differences. The FreeBSD VFS layer chooses to re-dirty pages when there is a failure (except when the device is removed) [6] while Linux hands over the failure handling responsibility to the individual file systems below the VFS layer (§3.3.4). *We hope that the Linux file-system maintainers will adopt a similar approach in an effort to handle `fsync` failures uniformly across file systems.* Note that it is also important to think about when to classify whether a device has been removed. For example, while storage devices connected over a network aren't really as permanent as local hard disks, they are more permanent than removable USB sticks. Temporary disconnects over a network need not be perceived as device removal and re-attachment; pages associated with such a device can be re-dirtied on write failure.

#6: Application developers do not target specific file systems. We observe that data-intensive applications configure their durability and error-handling strategies according to the OS they are running on, but treat all file systems on a specific operating system equally. Thus, as seen in Table 2, a single application can manifest different errors depending on the file system. *If the POSIX standard is not refined, applications may wish to handle `fsync` failures on different file systems differently.* Alternatively, applications may choose to code against *failure handling characteristics* as opposed to specific file systems, but this requires file systems to expose some interface to query characteristics such as "Post Failure Page State/Content" and "Immediate/Delayed Error Reporting".

#7: Applications employ a variety of strategies when fsync fails, but none are sufficient. As seen in Section 4.3, Redis chooses to trust the file system and does not even check `fsync` return codes, LMDB, LevelDB, and SQLite revert in-memory state and report the error to the application while PostgreSQL chooses to crash. We have seen that none of the applications retry `fsync` on failure; application developers appear to be aware that pages are marked clean on `fsync` failure and another `fsync` will not flush additional data to disk. Despite the fact that applications take great care to handle a range of errors from the storage stack (e.g., LevelDB writes CRC Checksums to detect invalid log entries and SQLite updates the header of the rollback journal only after the data is persisted to it), data durability cannot be guaranteed as long as `fsync` errors are not handled correctly. *While no one strategy is always effective, the approach currently taken by PostgreSQL to use direct IO may best handle `fsync` failures.* If file systems do choose to report failure handling characteristics in a standard format, applications may be able to employ better strategies. For example, applications can choose to keep track of dirtied pages and re-dirty them by reading and writing back a single byte if they know that the page content is not reverted on failure (ext4, XFS). On Btrfs, one would have to keep track of the page as well as its content. For applications that access multiple files, it is important to note that the files can exist on different file systems.

#8: Applications run recovery logic that accesses incorrect data in the page cache. Applications that depend on the page cache for faster recovery are susceptible to FalseFailures. As seen in LevelDB, SQLite, and PostgreSQL, when the *wal* incurs an `fsync` failure, the applications fail the operation and notify the user; In these cases, while the on-disk state may be corrupt, the entry in the page cache is valid; thus, an application that recovers state from the *wal* might read partially valid entries from the page cache and incorrectly update on-disk state. *Applications should read the on-disk content of files when performing recovery.*

#9: Application recovery logic is not tested with low level block faults. Applications test recovery logic and possibilities of data loss by either mocking system call return codes or emulating crash-restart scenarios, limiting interaction with the underlying file system. As a result, failure handling logic by the file system is not exercised. *Applications should test recovery logic using low-level block injectors that force underlying file-system error handling.* Alternatively, they could use a fault injector like CuttleFS that mimics different file-system error-handling characteristics.

6 Related Work

In this section, we discuss how our work builds upon and differs from past studies in key ways. We include works that study file systems through fault injection, error handling in file systems, and the impact of file-system faults on applications.

Our study on how file systems react to failures is related to work done by Prabhakaran et al. with IRON file systems [49] and a more recent study conducted by Jaffer et al. [40]. Other works study specific file systems such as NTFS [28] and ZFS [58]. All these studies inject failures beneath the file system and analyze if and how file systems detect and recover from them. These studies use system-call workloads (e.g., writes and reads) that make the file system interact with the underlying device.

While prior studies do exercise some portions of the `fsync` path through single system-call operations, they do not exercise the checkpoint path. More importantly, in contrast to these past efforts, our work focuses specifically on the *in-memory* state of a file system and the effects of *future operations* on a file system that has encountered a write fault. Specifically, in our work, we choose workloads that continue after a fault has been introduced. Such workloads help in understanding the after-effects of failures during `fsync` such as masking of errors by future operations, fixing the fault, or exacerbating it.

Mohan et al. [45] use bounded black-box crash testing to exhaustively generate workloads and discover many crash-consistency bugs by simulating power failures at different persistence points. Our work focuses on transient failures that may not necessarily cause a file system to crash and the effect on applications even though a file system may be consistent. Additionally, we inject faults in the middle of an `fsync` as opposed to after a successful `fsync` (persistence point).

Gunawi et al. describe the problem of failed intentions [36] in journaling file systems and suggest chained transactions to handle such faults during checkpointing. Another work develops a static-analysis technique named Error Detection and Propagation [37] and conclude that file systems neglect many write errors. Even though the Linux kernel has improved its block-layer error handling [10], file systems may still neglect write errors. Our results are purely based on injecting errors in bio requests that the file system can detect.

Vondra describes how certain assumptions about `fsync` behavior led to data loss in PostgreSQL [55]. The data loss behavior was reproduced using a device mapper with the dm-error target which inspired us to build our own fault injector (dm-loki [4]) atop the device mapper, similar to dm-inject [40]. Additionally, the FSQA suite (xfstests) [7] emulates write errors using the dm-flakey target [5]. While dm-flakey is useful for fault-injection testing, faults are injected based on current time; the device is available for x seconds and then exhibits unreliable behavior for y seconds (x and y being configurable). Furthermore, any change in configuration requires suspending the device. To increase determinism and avoid relying on time, dm-loki injects faults based on access patterns (e.g., fail the 2nd and 4th write to block 20) and is capable of accepting configuration changes without device suspension.

Recent work has shifted the focus to study the effects of file-system faults in distributed storage systems [34] and high-performance parallel systems [29]. Similarly, our work focuses on understanding how file systems and applications running on top of them behave in the presence of failures.

7 Conclusions

We show that file systems behave differently on `fsync` failure. Application developers can only assume that the underlying file system experienced a fault and that data may have either been persisted partially, completely, or not at all. We show that applications assuming more than the above are susceptible to data loss and corruptions. The widely perceived crash-restart fix in the face of `fsync` failures does not always work; applications recover incorrectly due to on-disk and in-memory mismatches.

However, we believe that applications can provide stronger guarantees if file systems are more uniform in their failure handling and error reporting strategies. Applications that care about durability should include sector- or block-level fault-injection tests to effectively test recovery code paths. Alternatively, such applications can choose to use CuttleFS to inject faults and mimic file system failure reactions.

We have open sourced CuttleFS at `https://github.com/WiscADSL/cuttlefs` along with the device-mapper kernel module and experiments to reproduce the results in this paper.

8 Acknowledgements

We thank Peter Macko (our shepherd), the anonymous reviewers of ATC '20, and the members of ADSL for their insightful comments and suggestions. We thank CloudLab [32] for providing a great environment to run our experiments. We also thank our sponsors: VMWare, NetApp, and Intel, for their generous support. This material was also supported by funding from NSF grants CNS-1421033, CNS-1763810 and CNS-1838733, and DOE grant DE-SC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DOE, or any other institutions.

References

[1] Atomic Commit In SQLite. https://www.sqlite.org/atomiccommit.html.

[2] Bug-207729 Mounting EXT4 with data_err=abort does not abort journal on data block write failure. https://bugzilla.kernel.org/show_bug.cgi?id=207729.

[3] Bug-27805553 HARD ERROR SHOULD BE REPORTED WHEN FSYNC() RETURN EIO. https://github.com/mysql/mysql-server/commit/8590c8e12a3374eeccb547359750a9d2a128fa6a.

[4] Custom Fault Injection Device Mapper Target: dm-loki. https://github.com/WiscADSL/dm-loki.

[5] Device Mapper: dm-flakey. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-flakey.html.

[6] FreeBSD VFS Layer re-dirties pages after failed block write. https://github.com/freebsd/freebsd/blob/0209fe3398be56e5e042c422a96a4fbc654247f4/sys/kern/vfs_bio.c#L2646.

[7] FSQA (xfstests). https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/about/.

[8] Fsync Errors - PostgreSQL wiki. https://wiki.postgresql.org/wiki/Fsync_Errors.

[9] fsync(2) - Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/fdatasync.2.html.

[10] Improved block-layer error handling. https://lwn.net/Articles/724307/.

[11] Is data=journal safer for Ext4 as opposed to data=ordered? https://unix.stackexchange.com/q/127235.

[12] LevelDB. https://github.com/google/leveldb.

[13] Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.

[14] POSIX Specification for fsync. https://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html.

[15] PostgreSQL. https://www.postgresql.org/.

[16] PostgreSQL: Write-Ahead Logging (WAL). https://www.postgresql.org/docs/current/wal-intro.html.

[17] PostgreSQL's handling of fsync() errors is unsafe and risks data loss at least on XFS . https://www.postgresql.org/message-id/flat/CAMsr%2BYHh%2B5Oq4xziwwoEfhoTZgr07vdGG%2Bhu%3D1adXx59aTeaoQ%40mail.gmail.com.

[18] Redis. https://redis.io/.

[19] Redis Persistence. https://redis.io/topics/persistence.

[20] SQLite. https://www.sqlite.org/index.html.

[21] SQLite Write-Ahead Logging. https://www.sqlite.org/wal.html.

[22] SystemTap. https://sourceware.org/systemtap/.

[23] Why does ext4 clear the dirty bit on I/O error? https://www.postgresql.org/message-id/edc2e4d5-5446-e0db-25da-66db6c020cc3%40commandprompt.com.

[24] WT-4045 Don't retry fsync calls after EIO failure. https://github.com/wiredtiger/wiredtiger/commit/ae8bccce3d8a8248afa0e4e0cf67674a43dede96.

[25] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[26] Lakshmi N. Bairavasundaram, Garth Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, CA, February 2008.

[27] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289–300, San Diego, CA, June 2007.

[28] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, pages 502–511, Anchorage, Alaska, June 2008.

[29] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 1–11, Beijing, China, June 2018.

[30] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 228–243, Farmington, PA, November 2013.

[31] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, CA, February 2012.

[32] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019.

[33] Christian Forfang. Evaluation of High Performance Key-Value Stores. Master's thesis, Norwegian University of Science and Technology, June 2014.

[34] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 149–165, Santa Clara, CA, February 2017.

[35] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, CA, November 1994.

[36] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 293–306, Stevenson, WA, October 2007.

[37] Haryadi S. Gunawi, Cindy Rubio-González, Remzi H. Arpaci-Dusseau Andrea C. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 207–222, San Jose, CA, February 2008.

[38] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 155–162, Austin, Texas, November 1987.

[39] FUSE (Filesystem in Userspace). The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.

[40] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating File System Reliability on Solid State Drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–797, Renton, WA, July 2019.

[41] Hannu H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.

[42] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, CA, February 2008.

[43] Avantika Mathur, Mingming Cao, and Andreas Dilger. Ext4: The Next Generation of the Ext3 File System. *Usenix Association*, 32(3):25–30, June 2007.

[44] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, pages 99–111, Boston, MA, June 1994.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 33–50, Carlsbad, CA, October 2018.

[46] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 181–196, Santa Clara, CA, February 2017.

[47] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 433–448, Broomfield, CO, October 2014.

[48] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 802–811, Yokohama, Japan, June 2005.

[49] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, UK, October 2005.

[50] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, August 2013.

[51] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, pages 71–84, San Jose, CA, February 2010.

[52] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–323, Washington, D.C., January 1990.

[53] Chuck Silvers. UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*, pages 285–290, San Diego, CA, June 2000.

[54] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

[55] Tomas Vondra. PostgreSQL vs. fsync. How is it possible that PostgreSQL used fsync incorrectly for 20 years, and what we'll do about it. Brussels, Belgium, February 2019. https://archive.fosdem.org/2019/schedule/event/postgresql_fsync/.

[56] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 211–226, Oakland, CA, February 2018.

[57] Yiying Zhang and Steven Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST '15)*, pages 1–10, Santa Clara, CA, May 2015.

[58] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, pages 29–42, San Jose, CA, February 2010.

DupHunter: Flexible High-Performance Deduplication for Docker Registries

Nannan Zhao[1], Hadeel Albahar[1], Subil Abraham[1], Keren Chen[1], Vasily Tarasov[2],
Dimitrios Skourtis[2], Lukas Rupprecht[2], Ali Anwar[2], and Ali R. Butt[1]
[1]*Virginia Tech* [2]*IBM Research—Almaden*

Abstract

The rise of containers has led to a broad proliferation of container images. The associated storage performance and capacity requirements place high pressure on the infrastructure of container registries that store and serve images. Exploiting the high file redundancy in real-world container images is a promising approach to drastically reduce the demanding storage requirements of the growing registries. However, existing deduplication techniques significantly degrade the performance of registries because of the high layer restore overhead.

We propose DupHunter, a new Docker registry architecture, which not only natively deduplicates layers for space savings but also reduces layer restore overhead. DupHunter supports several configurable *deduplication modes*, which provide different levels of storage efficiency, durability, and performance, to support a range of uses. To mitigate the negative impact of deduplication on the image download times, DupHunter introduces a *two-tier storage hierarchy* with a novel layer prefetch/preconstruct cache algorithm based on user access patterns. Under real workloads, in the *highest data reduction mode*, DupHunter reduces storage space by up to 6.9× compared to the current implementations. In the *highest performance mode*, DupHunter can reduce the `GET` layer latency up to 2.8× compared to the state of the art.

1 Introduction

Containerization frameworks such as Docker [2] have seen a remarkable adoption in modern cloud environments. This is due to their lower overhead compared to virtual machines [7,38], a rich ecosystem that eases application development, deployment, and management [17], and the growing popularity of microservices [69]. By now, all major cloud platforms endorse containers as a core deployment technology [10,28,31,47]. For example, Datadog reports that in 2018, about 21% of its customers' monitored hosts ran Docker and that this trend continues to grow by about 5% annually [19].

Container images are at the core of containerized applications. An application's container image includes the executable of the application along with a complete set of its dependencies—other executables, libraries, and configuration and data files required by the application. Images are structured in *layers*. When building an image with Docker, each executed command, such as `apt install`, creates a new layer on top of the previous one [4], which contains the files that the command has modified or added. Docker leverages union file systems [64] to efficiently merge layers into a single file system tree when starting a container. Containers can share identical layers across different images.

To store and distribute container images, Docker relies on image *registries* (e.g., Docker Hub [3]). Docker clients can push images to or pull them from the registries as needed. On the registry side, each layer is stored as a compressed tarball and identified by a content-based address. The Docker registry supports various storage backends for saving and retrieving layers. For example, a typical large-scale setup stores each layer as an object in an object store [32,51].

As the container market continues to expand, Docker registries have to manage a growing number of images and layers. Some conservative estimates show that in spring 2019, Docker Hub alone stored at least 2 million *public* images totaling roughly 1 PB in size [59,72]. We believe that this is just the tip of the iceberg and the number of *private* images is significantly higher. Other popular public registries [9, 27, 35, 46], as well as on-premises registry deployments in large organizations, experience a similar surge in the number of images. As a result, organizations spend an increasing amount of their storage and networking infrastructure on operating image registries.

The storage demand for container images is worsened by the large amount of duplicate data in images. As Docker images must be self-contained by definition, different images frequently include the same, common dependencies (e.g., libraries). As a result, different images are prone to contain a high number of duplicate files as shared components exist in more than one image.

To reduce this redundancy, Docker employs layer sharing. However, this is insufficient as layers are coarse and rarely identical because they are built by developers independently and without coordination. Indeed, a recent analysis of the Docker Hub image dataset showed that about 97% of files across layers are duplicates [72]. Registry storage backends exacerbate the redundancy further due to the replication they perform to improve image durability and availability [12].

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [52]. However, applying deduplication to a Docker registry is challenging due to two main reasons: 1) layers are stored in the registry as **compressed** tarballs that do not deduplicate well [44]; and 2) decompressing layers first and storing individual files incurs high reconstruction overhead and slows down image pulls. The slowdowns during image pulls are especially harmful because they contribute directly to the startup times of containers. Our experiments show that, on average, naive deduplication increases layer pull latencies by up to 98× compared to a registry without deduplication.

In this paper, we propose DupHunter, the first Docker registry that natively supports deduplication. DupHunter is designed to increase storage efficiency via layer deduplication while reducing the corresponding layer restoring overhead. It utilizes domain-specific knowledge about the stored data and the storage system to reduce the impact of layer deduplication on performance. For this purpose, DupHunter offers five key contributions:

1. DupHunter exploits existing replication to improve performance. It keeps a specified number of layer replicas as-is, without decompressing and deduplicating them. Accesses to these replicas do not experience layer restoring overhead. Any additional layer replicas needed to guarantee the desired availability are decompressed and deduplicated.
2. DupHunter deduplicates rarely accessed layers more aggressively than popular ones to speed up accesses to popular layers and achieve higher storage savings.
3. DupHunter monitors user access patterns and proactively restores layers *before* layer download requests arrive. This allows it to avoid reconstruction latency during pulls.
4. DupHunter groups files from a single layer in *slices* and evenly distributes the slices across the cluster, to parallelize and speed up layer reconstruction.
5. We use DupHunter to provide the first comprehensive analysis of the impact of different deduplication levels (file and block) and redundancy policies (replication and erasure coding) on registry performance and space savings.

We evaluate DupHunter on a 6-node cluster using real-world workloads and layers. In the *highest performance mode*, DupHunter outperforms the state-of-the-art Docker registry, Bolt [41], by reducing layer pull latencies by up to 2.8×. In the *highest deduplication mode*, DupHunter reduces storage consumption by up to 6.9×. DupHunter also supports other deduplication modes that support various trade-offs in performance and space savings.

2 Background and Related Work

We first provide the background on the Docker registry and then discuss existing deduplication works.

2.1 Docker Registry

The main purpose of a Docker registry is to store and distribute container images to Docker clients. A registry provides a REST API for Docker clients to *push* images to and *pull* images from the registry [20, 21]. Docker registries group images into *repositories*, each containing versions (*tags*) of the same image, identified as `<repo-name:tag>`. For each tagged image in a repository, the Docker registry stores a *manifest*, i.e., a JSON file that contains the runtime configuration for a container image (e.g., environment variables) and the list of layers that make up the image. A layer is stored as a compressed archival file and identified using a digest (SHA-256) computed over the uncompressed contents of the layer. When pulling an image, a Docker client first downloads the manifest and then the referenced layers (that are not already present on the client). When pushing an image, a Docker client first uploads the layers (if not already present in the registry) and then the manifest.

The current Docker registry software is a single-node application with a RESTful API. The registry delegates storage to a backend storage system through corresponding storage drivers. The backend storage can range from local file systems to distributed object storage systems such as Swift [51] or others [1, 5, 32, 51]. To scale the registry, organizations typically deploy a load balancer or proxy in front of several independent registry instances [11]. In this case, client requests are forwarded to the destination registries through a proxy, then served by the registries' backend storage system. To reduce the communication overhead between the proxy, registry, and backend storage system, Bolt [41] proposes to use a consistent hashing function instead of a proxy, distribute requests to registries, and utilize the local file system on each registry node to store data instead of using a remote distributed object storage system. Multiple layer replicas are stored on Bolt registries for high availability and reliability. DupHunter is implemented based on the architecture of Bolt registry for high scalability.

Registry performance is critical to Docker clients. In particular, the layer `pulling` performance (i.e., `GET` layer performance) impacts container startup times significantly [30]. A number of works have studied various dimensions of registry performance for a Docker image dataset [11, 14, 30, 60, 64, 71, 72]. However, such works do not provide deduplication for the registry. A community proposal exists to add file-level deduplication to container images [8], but as of now lacks even a detailed design, let alone performance analysis. Skourtis et al. [59] propose restructuring layers to optimize for various dimensions, including registry storage utilization. Their approach does not remove all duplicates, whereas DupHunter leaves images unchanged and can eliminate all duplicates in the registry. Finally, a lot of works aim to reduce the size of a single container image [22,29,54,65], and are complementary to DupHunter.

2.2 Deduplication

Data deduplication has received considerable attention, particularly for virtual machine images [33, 36, 61, 73]. Many deduplication studies focus on primary and backup data deduplication [23–25,39,40,42,48,58,63,68,74] and show the effectiveness of file- and block-level deduplication [45, 62]. To further reduce storage space, integrating block-level deduplication with compression has been proposed [66]. In addition to local deduplication schemes, a global deduplication method [49] has also been proposed to improve the deduplication ratio and provide high scalability for distributed storage systems.

Data restoring latency is an important factor for storage systems with deduplication support. Efficient chunk caching algorithms and forward assembly are proposed to accelerate data restore performance [15]. At first glance, one could apply existing deduplication techniques to solve the issue of high data redundancy among container images. However, as we demonstrate in detail in §3.2, such a naive approach leads to slow reconstruction of layers on image pulls, which severely degrades container startup times. DupHunter is specifically designed for Docker registries, which allows it to leverage image and workload information to reduce deduplication and layer restore overhead.

3 Motivating Observations

The need and feasibility of DupHunter is based on three key observations: 1) container images have a lot of redundancy; 2) existing scalable deduplication technologies significantly increase image pull latencies; and 3) image access patterns can be predicted reliably.

3.1 Redundancy in Container Images

Container image layers exhibit a large degree of redundancy in terms of duplicate files. Although Docker supports the sharing of layers among different images to remove some redundant data in the Docker registry, this is not sufficient to effectively eliminate duplicates. According to the deduplication analysis of the Docker Hub dataset [72], 97% of files have more than one file duplicate, resulting in a deduplication ratio of 2× in terms of capacity. We believe that the deduplication ratio is much higher when private repositories are taken into account.

Table 1: Dedup. ratio vs. increase in GET layer latency.

Technology	Dedup ratio, compressed layers	Dedup ratio, uncompressed layers	GET latency increase wrt. uncompressed layers
Jdupes	1	2.1	36 ×
VDO	1	4	60 ×
Btrfs	1	2.3	51 ×
ZFS	1	2.3	50 ×
Ceph	1	3.1	98 ×

The duplicate files are executables, object code, libraries, and source code, and are likely imported by different image developers using package installers or version control systems such as `apt`, `pip`, or `git` to install similar dependencies. However, as layers often share many but not all files, this redundancy cannot be eliminated by Docker's current layer sharing approach.

R-way replication for reliability further fuels the high storage demands of Docker registries. Hence, satisfying demand by adding more disks and scaling out storage systems quickly becomes expensive.

3.2 Drawbacks of Existing Technologies

A naive approach to eliminating duplicates in container images could be to apply an existing deduplication technique. To experimentally demonstrate that such a strategy has significant shortcomings, we try four popular local deduplication technologies, `VDO` [67], `Btrfs` [13], `ZFS` [70], `Jdupes` [34], in a single-node setup and on one distributed solution, `Ceph` [16], on a 3-node cluster. The deduplication block sizes are set to 4KB for both `VDO` and `Ceph`, and 128KB for both `Btrfs` [13] and `ZFS` [70] by default. Table 1 presents the deduplication ratio and pull latency overhead for each technology in two cases: 1) when layers are stored compressed (as-is); and 2) when layers are uncompressed and unpacked into their individual files. Note that the deduplication ratios are calculated against the case when all layers are compressed (the details of the dataset and testbed are presented in §6).

Deduplication ratios. Putting the original compressed layer tarballs in any of the deduplication systems results, unsuprisingly, in a deduplication ratio of 1. This is because even a single byte change in any file in a tarball scrambles the content of the compressed tarball entirely [18, 44]. Hence, to expose the redundancy to the deduplication systems, we decompress every layer before storing it.

After decompression, all deduplication schemes yield significant deduplication ratios. `Jdupes`, `Btrfs`, and `ZFS` reduce the dataset to about half and achieve deduplication ratios of 2.1, 2.3, and 2.3, respectively. `Ceph` has a higher

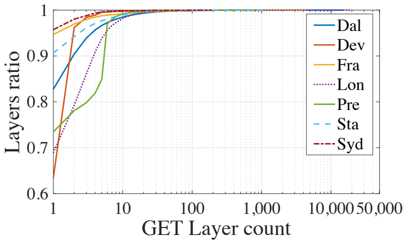


Figure 1: CDF of `GET` layer request count.

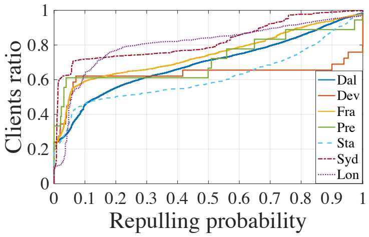


Figure 2: CDF of client repulling probability.

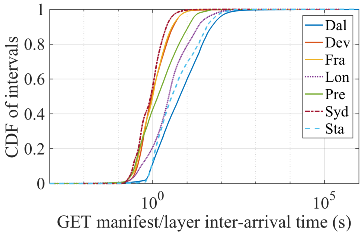


Figure 3: CDF of `GET` manifest/layer inter-arrival time.

deduplication ratio since it uses a smaller deduplication block size, while `VDO` shows the highest deduplication ratio as it also compresses deduplicated data.

It is important to note that for an enterprise-scale registry, a large number of storage servers need to be deployed and single-node deduplication systems (`Jdupes`, `Btrfs`, `ZFS`, and `VDO`) can only deduplicate data within a single node. Therefore, in a multi-node setup, such solutions can never achieve optimal *global* deduplication, i.e., duplication across nodes.

Pull latencies. To analyze layer pull latencies, we implement a layer restoring process for each technology. Restoring includes fetching files, creating a layer tarball, and compressing it. We measure the average `GET` layer latency and calculate the restore overhead compared to `GET` requests without layer deduplication.

As shown in Table 1, the restoration overhead is high. The file-level deduplication scheme `Jdupes` increases the `GET` layer latency by 36×. This is caused by the expensive restoring process. `Btrfs`, `ZFS`, and `VDO` show an increase of more than 50×, as they are block-level deduplication systems, and hence they also add file restoring overhead. The overhead for `Ceph` is the highest because restoration is distributed and incurs network communication.

In summary, our analysis shows that while existing technologies can provide storage space savings for container images (after decompression), they incur high cost during image pulls due to slow layer reconstruction. At the same time, pull latency constitutes the major portion of container startup times even without deduplication. According to [30], pulling images accounts for 76% of container startup times. This means that, for example, for Btrfs the increase of layer GET latency by 51× would prolong container startup times by 38×. Hence, deduplication has a major negative impact on the startup times of containerized applications.

3.3 Predictable User Access Patterns

A promising approach to mitigate layer restoring overhead is predicting which layers will be accessed and preconstruct them. In DupHunter, we can exploit the fact that when a Docker client pulls an image from the registry, it first retrieves the image manifest, which includes references to the image layers.

User pulling patterns. Typically, if a layer is already stored locally, then the client will not fetch this layer again. However, higher-level container orchestrators allow users to configure different policies for starting new containers. For example, Kubernetes allows policies such as `IfNotPresent`, i.e., only get the layer if it has not been pulled already, or `AlwaysGet`, i.e., always retrieve the layer, even if it is already present locally. These different behaviors need to be considered when predicting whether a layer will be pulled by a user or not.

We use the IBM Cloud registry workload [11] to analyze the likelihood for a user to *repull* an already present layer. The traces span ∼80 days for 7 registry clusters: Dallas, Frankfurt, London, Sydney, Development, Prestaging, and Staging. Figure 1 shows the CDF of layer `GET` counts by the same clients. The analysis shows that the majority of layers are only fetched once by the same clients. For example, 97% of layers from `Syd` are only fetched once by the same clients. However, there are clients that pull the same layers repeatedly. E.g., a client from London fetched the same layer 19,300 times.

Figure 2 shows the corresponding client repull probability, calculated as the number of repulled layers divided by the number of total `GET` layer requests issued by the same client. We see that 50% of the clients have a repull probability of less than 0.2 across all registries. We also observe that the slope of the CDFs is steep at both lower and higher probabilities, but becomes flat in the middle. This suggests that, by observing access patterns, we are able to classify clients into two categories, always-pull clients and pull-once clients, and predict, whether they will pull a layer or not by keeping track of user access history.

Layer preconstruction. We analyze the inter-arrival time between a `GET` manifest request and the subsequent `GET` layer request. As shown in Figure 3, the majority of intervals are greater than 1 second. For example, 80% of intervals from London are greater than 1 second, and 60% of the intervals from Sydney are greater than 5 seconds.

There are several reasons for this long gap. First, when

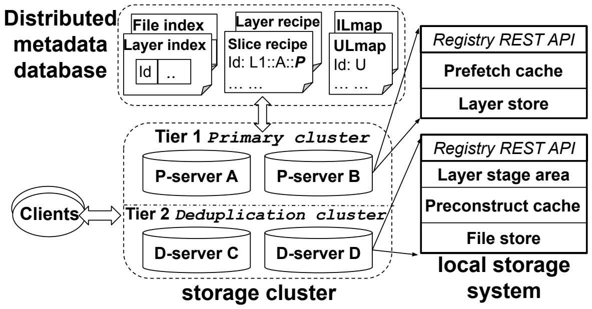


Figure 4: DupHunter architecture.

fetching an image from a registry, the Docker client fetches a fixed number of layers in parallel (three by default) starting from the lowest layer. In the case where an image contains more than three layers, the upper layers have to wait until the lower layers are downloaded, which delays the `GET` layer request for these layers. Second, network delay between clients and registry often accounts for a large portion of the `GET` latency in cloud environments.

As we show in §6, layer preconstruction can significantly reduce layer restoring overhead. In the case of a shorter duration between a `GET` manifest request and its subsequent `GET` layer requests, layer preconstruction can still be beneficial because the layer construction starts prior to the arrival of the `GET` request.

4 DupHunter Design

In this section, we first provide an overview of DupHunter (§4.1). We then describe in detail how it deduplicates (§4.2) and restores (§4.3) layers, and how it further improves performance via predictive cache management (§4.4). Finally, we discuss the integration of sub-file deduplication and erasure coding with DupHunter (§4.5).

4.1 Overview

Figure 4 shows the architecture of DupHunter. DupHunter consists of two main components: 1) a cluster of *storage servers*, each exposing the registry REST API; and 2) a distributed *metadata database*. When uploading or downloading layers, Docker clients communicate with any DupHunter server using the registry API. Each server in the cluster contains an API service and a backend storage system. The backend storage systems store layers and perform deduplication, keeping the deduplication metadata in the database. DupHunter uses three techniques to reduce deduplication and restoring overhead: 1) replica deduplication modes; 2) parallel layer reconstruction; and 3) proactive layer prefetching/preconstruction.

Replica deduplication modes. For higher fault tolerance and availability, existing registry setups replicate layers. DupHunter also performs layer replication, but additionally deduplicates files inside the replicas.

A *basic deduplication mode* n (B-mode n) defines that DupHunter should only keep n layer replicas intact and deduplicate the remaining $R-n$ layer replicas, where R is the layer replication level. At one extreme, B-mode R means that no replicas should be deduplicated, and hence provides the best performance but no data reduction. At the other end, B-mode 0 deduplicates all layer replicas, i.e., it provides the highest deduplication ratio but adds restoration overhead for `GET` requests. The remaining in-between B-modes allow to trade off performance for data reduction.

For heavily skewed workloads, DupHunter also provides a *selective deduplication mode* (S-mode). The S-mode utilizes the skewness in layer popularity, observed in [11], to decide how many replicas should be deduplicated for each layer. As there are hot layers that are pulled frequently, S-mode sets the number of intact replicas proportional to their popularity. This means that hot layers have more intact replicas, and hence can be served faster, while cold layers are deduplicated more aggressively.

Deduplication in DupHunter, for the example of B-mode 1, works as follows: DupHunter first creates 3 layer replicas across 3 servers. It keeps a single layer replica as the *primary layer replica* on one server. Deduplication is then carried out in one of the other servers storing a replica, i.e., the layer replica is decompressed and any duplicate files are discarded while unique files are kept. The unique files are replicated and saved on different servers for fault tolerance. Once deduplication is complete, the remaining two layer replicas are removed. Any subsequent `GET` layer requests are sent to the primary replica server first since it stores the complete layer replica. If that server crashes, one of the other servers is used to rebuild the layer and serve the `GET` request.

To support different deduplication modes, DupHunter stores a mix of both layer tarballs and individual files. This makes data placement decision more complex with respect to fault tolerance because individual files and their corresponding layer tarballs need to be placed on different servers. As more tarballs and files are stored in the cluster, the placement problem gets more challenging.

To avoid accidentally co-locating layer tarballs and unique files, which are present in the tarball, and simplify the placement problem, DupHunter divides storage servers into two groups (Figure 4): a *primary cluster* consisting of *P-servers* and a *deduplication cluster* consisting of *D-servers*. P-servers are responsible for storing full layer tarball replicas and replicas of the manifest, while D-servers deduplicate, store, and replicate the unique files from the layer tarballs. The split allows DupHunter to treat layers and individual files separately and prevent co-location during placement.

P- and D-servers form a 2-tier storage hierarchy. In the default case, the primary cluster serves all incoming `GET` requests. If a request cannot be served from the primary cluster (e.g., due to a node failure, or DupHunter operating in B-mode 0 or S-mode), it will be forwarded to the deduplication cluster and the requested layer will be reconstructed.

Parallel layer reconstruction. DupHunter speeds up layer reconstruction through parallelism. As shown in Figure 4, each D-server's local storage is divided into three parts: the layer stage area, preconstruction cache, and file store. The layer stage area temporarily stores newly added layer replicas. After deduplicating a replica, the resulting unique files are stored in a content addressable file store and replicated to the peer servers to provide redundancy. Once all file replicas have been stored, the layer replica is deleted from the layer stage area.

DupHunter distributes the layer's unique files onto several servers (see §4.2). All files on a single server belonging to the same layer are called a *slice*. A slice has a corresponding *slice recipe*, which defines the files that are part of this slice, and a *layer recipe* defines the slices needed to reconstruct the layer. This information is stored in DupHunter's metadata database. This allows D-servers to rebuild layer slices in parallel and thereby improve reconstruction performance. DupHunter maintains layer and file fingerprint indices in the metadata database.

Predictive cache prefetch and preconstruction. To improve the layer access latency, DupHunter employs a cache layer in both the primary and the deduplication clusters, respectively. Each P-server has an in-memory *user-behavior based prefetch cache* to reduce disk I/Os. When a `GET` manifest request is received from a user, DupHunter predicts which layers in the image will actually need to be `pulled` and prefetches them in the cache. Additionally, to reduce layer restoring overhead, each D-server maintains an on-disk *user-behavior based preconstruct cache*. As with the prefetch cache, when a `GET` manifest request is received, DupHunter predicts which layers in the image will be `pulled`, preconstructs the layers, and loads them in the preconstruct cache. To accurately predict which layers to prefetch, DupHunter maintains two maps: *ILmap* and *ULmap*. ILmap stores the mapping between images and layers while ULmap keeps track of a user's access history, i.e., which layers the user has pulled and how many times (see §4.4).

4.2 Deduplicating Layers

As in the traditional Docker registry, DupHunter maintains a *layer index*. After receiving a `PUT` layer request, DupHunter first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. If not, DupHunter, replicates the layer r times across the P-servers and submits the remaining $R - r$ layer replicas to the D-servers. Those replicas are temporarily stored in the layer stage areas of the D-servers. Once the replicas have been stored successfully, DupHunter notifies the client of the request completion.

File-level deduplication. Once in the staging area, one of the D-servers decompresses the layer and starts the deduplication process. First, it extracts file entries from the tar archive. Each file entry is represented as a *file header* and the associated *file content* [26]. The file header contains metadata such as file name, path, size, mode, and owner information. DupHunter records every file header in slice recipes (described below) to be able to correctly restore the complete layer archive later.

To deduplicate a file, DupHunter computes a file Id by hashing the file content and checks if the Id is already present in the file index. If present, the file content is discarded. Otherwise, the file content is assigned to a D-server and stored in its file store, and the file Id is recorded in the file index. The file index maps different file Ids to their physical replicas stored on different D-servers.

Layer partitioning. DupHunter picks D-servers for files to improve reconstruction times. For that, it is important that different layer slices are similarly sized and evenly distributed across D-servers. To achieve this, DupHunter employs a greedy packing algorithm. Consider first the simpler case in which each file only has a single replica. DupHunter first computes the total size of the layer's existing shared files on each D-server (this might be 0 if a D-server does not store any shared files for the layer). Next, it assigns the largest new unique file to the smallest partition until all the unique files are assigned. Note that during layer partitioning, DupHunter does not migrate existing shared files to reduce I/O overhead.

In the case where a file has more than one replica, DupHunter performs the above-described partitioning *per replica*. That means that it first assigns the primary replicas of the new unique files to D-servers according to the location of the primary replicas of the existing shared files. It then does the same for the secondary replicas and so on. DupHunter also ensures that two replicas of the same file are never placed on the same node.

Unique file replication. Next, DupHunter replicates and distributes the unique file replicas across D-servers based on the layer partitioning. The headers and content pointers of all files in the deduplicated layer that are assigned to a specific D-server are included in that D-server's *slice recipe* for that layer. After file replication, DupHunter adds the new slice recipes to the metadata database.

DupHunter also creates a *layer recipe* for the uploaded layer and stores it in the metadata database. The layer recipe records all the D-servers that store slices for that layer and which can act as *restoring workers*. When a layer needs to be reconstructed, one worker is selected as the *restoring master*, responsible for gathering all slices

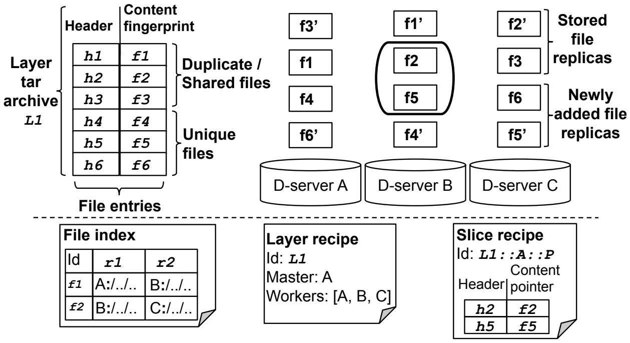


Figure 5: Layer dedup., replication, and partitioning.

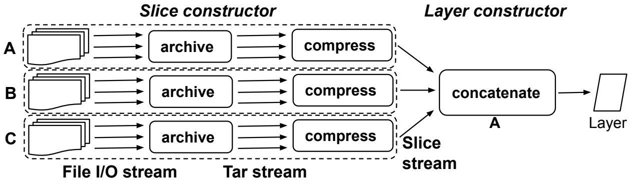


Figure 6: Parallel streaming layer construction.

and rebuilding the layer (see §4.3).

Figure 5 shows an example deduplication process. The example assumes B-mode 1 with 3-way replication, i.e., each unique file has two replicas distributed on two different D-servers. The files $f1$, $f2$, and $f3$ are already stored in DupHunter, and $f1'$, $f2'$, and $f3'$ are their corresponding replicas. Layer $L1$ is being pushed and contains files $f1$–$f6$. $f1$, $f2$, and $f3$ are *shared files* between $L1$ and other layers, and hence are discarded during file-level deduplication. The unique files $f4$, $f5$ and $f6$ are added to the system and replicated to D-servers A, B, and C.

After replication, server B contains $f2$, $f5$, $f1'$, and $f4'$. Together $f2$ and $f5$ form the *primary slice* of $L1$, denoted as $L1 :: B :: P$. This slice Id contains the layer Id the slices belongs to ($L1$), the node, which stores the slice (B) and the backup level (P for primary). The two backup file replicas $f1'$ and $f4'$ on B form the *backup slice* $L1 :: B :: B$. During layer restoring, $L1$ can be restored by using any combination of primary and backup slices to achieve maximum parallelism.

4.3 Restoring Layers

The restoring process works in two phases: slice reconstruction and layer reconstruction. Considering the example in Figure 5, restoring works as follows:

According to $L1$'s layer recipe, the restoring workers are D-servers A, B, and C. The node with the largest slice is picked as the restoring master, also called *layer constructor* (A in the example). Since A is the restoring master it sends `GET slice` requests for the primary slices to B and C. If a primary slice is missing, the master locates its corresponding backup slice and sends a `GET slice` request to the corresponding D-server.

After a `GET slice` request has been received, B's and C's *slice constructors* start rebuilding their primary slices and send them to A as shown in Figure 6. Meanwhile, A instructs its local slice constructor to restore its primary slice for $L1$. To construct a layer slice, a slice constructor first gets the associated slice recipe from the metadata database. The recipe is keyed by a combination of layer Id, host address and requested backup level, e.g., $L1 :: A :: P$. Based on the recipe, the slice constructor creates a slice tar file by concatenating each file header and the corresponding file contents; it then compresses the slice and passes it to the master. The master concatenates all the compressed slices into a single compressed layer tarball and sends it back to the client.

The layer restoration performance is critical to keep `pull` latencies low. Hence, DupHunter parallelizes slice reconstruction on a single node and avoids generating intermediate files on disk to reduce disk I/O.

4.4 Caching and Preconstructing Layers

DupHunter maintains a cache layer in both the primary and deduplication clusters to speedup `pull` requests. The primary cluster cache (in-memory prefetch cache) is to avoid disk I/O during layer retrievals while the deduplication cluster on-disk cache stores preconstructed layers, which are likely to be accessed in the future. Both caches are filled based on the user access patterns seen in §3.

Request prediction. To accurately predict layers that will be accessed in the future, DupHunter keeps track of image metadata and user access patterns in two data structures: *ILmap* and *ULmap*. ILmap maps an image to its containing *layer set*. ULmap stores for each user the layers the user has accessed and the corresponding pull count. A user is uniquely identified by extracting the sender IP address from the request. If DupHunter has not seen an IP address before, it assumes that the request comes from a new host, which does not store any layers yet.

When a `GET manifest` request r is received, DupHunter first calculates a set of image layers that have not been pulled by the user *r.addr* by calculating the difference S_Δ between the image's layer set and the user's accessed layer set:

$$S_\Delta = ILmap[r.img] - ULmap[r.addr].$$

The layers in S_Δ are expected to be accessed soon.

Recall from §3.3 that some users *always* pull layers, no matter if the layers have been previously pulled. To detect such users, DupHunter maintains a *repull probability* γ for each user. For a `GET manifest` request r by a user

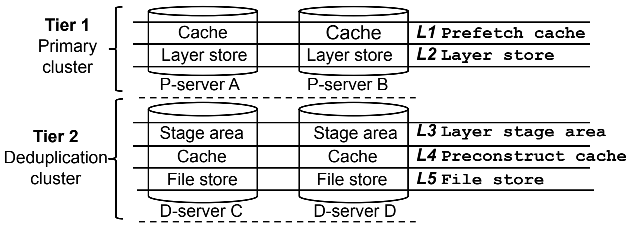


Figure 7: Tiered storage architecture.

$r.addr$, γ is computed as

$$\gamma[r.addr] = \sum_{l \in RL} l.pullCount \Big/ \sum_{l \in L} l.pullCount$$

where RL is the set of layers that the user has repulled before (i.e., with a pull count > 1) and L is the set of all layers the user has ever pulled. DupHunter updates the pull counts every time it receives a `GET layer` request.

DupHunter compares the clients' repull probability to a predefined threshold ε. If $\gamma[r.addr] > \varepsilon$, then DupHunter classifies the user as a repull user and computes the subset, $S_\cap$, of layers from the requested image that have already been pulled by the user:

$$S_\cap = ILmap[r.img] \cap ULmap[r.addr].$$

It then fetches the layers in $S_\cap$ into the cache.

Cache handling in tiered storage. The introduction of the two caches results in a 5-level 2-tier storage architecture of DupHunter as shown in Figure 7. Requests are passed through the tiers from top to bottom. Upon a `GET` layer request, DupHunter first determines the P-server(s) which is (are) responsible for the layer and searches the prefetch cache(s). If the layer is present, the request will be served from the cache. Otherwise, the request will be served from the layer store.

If a `GET layer` request cannot be served from the primary cluster due to a failure of the corresponding P-server(s), the request will be forwarded to the deduplication cluster. In that case, DupHunter will first lookup the layer recipe. If the recipe is not found, it means that the layer has not been fully deduplicated yet and DupHunter will serve the layer from one of the layer stage areas of the responsible D-servers. If the layer recipe is present, DupHunter will contact the restoring master to check, whether the layer is in its preconstruct cache. Otherwise, it will instruct the restoring master to rebuild the layer.

Both the prefetch and the preconstruct caches are write-through caches. When a layer is evicted, it is simply discarded since the layers are read-only. We use an Adaptive Replacement Cache (ARC) replacement policy [43], which keeps track of both the frequently and recently used layers and adapts to changing access patterns.

4.5 Discussion

The goal of DupHunter is to provide flexible deduplication modes to meet different space-saving and performance requirements and mitigate layer restore overhead. The above design of DupHunter mainly focuses on file-level deduplication and assumes layer replication.

To achieve a higher deduplication ratio, DupHunter can integrate with block-level deduplication. After removing redundant files, D-servers can further perform block-level deduplication only on unique files by using systems such as `VDO` [67] and `Ceph` [49]. However, higher deduplication ratios come with higher layer restoring overhead as the restoring latency for block-level deduplication is higher than that of file level as we show in §6. This is because to restore a layer, its associated files need to be first restored, which incurs extra overhead. Furthermore, when integrating with a global block-level deduplication scheme, the layer restoring overhead will be higher due to network communication. In this case, it is beneficial to maintain a number of layer replicas on P-servers to maintain a good performance.

While DupHunter exploits existing replication schemes, it is not limited to those. If the registry is using erasure coding for reliability, DupHunter can integrate with the erasure coding algorithm to improve space efficiency. Specifically, after removing redundant files from layers, DupHunter can store unique files as erasure-coded chunks. While DupHunter can not make use of existing replicas to improve pull performance in this case, its preconstruct cache remains beneficial to mitigate high restoring overheads as shown in §6.

A known side effect when performing deduplication is that the loss of a chunk has a bigger impact on fault tolerance as the chunk is referenced by several objects [57]. To provide adequate fault tolerance, DupHunter maintains at least three copies of a layer (either as full layer replicas or unique files that can rebuild the layer) in the cluster.

5 Implementation

We implemented DupHunter[1] in Go by adding $\sim 2,000$ lines of code to Bolt [41]. Note that Bolt is based on the reference Docker registry [20] for high availability and scalability (see Bolt details in §2.1.)

DupHunter can use any POSIX file system to store its data and uses Redis [6] for metadata, i.e., slice and layer recipes, file and layer indices, and ULmap and ILmap. We chose Redis because it provides high lookup and update performance and it is widely used in production systems. Another benefit of Redis is that it comes with a Go client library, which makes it easy to integrate with the Docker

[1] DupHunter's code is available at https://github.com/nnzhaocs/DupHunter.

Table 2: Workload parameters.

Trace	#GET Layer	#GET Manifest	#PUT Layer	#PUT Manifest	#Uniq. Layer	#Accessed Uniq. Dataset Size (GB)
Dal	6963	7561	453	23	1870	18
Fra	4117	10350	508	25	1012	9
Lon	2570	11808	582	40	1979	13
Syd	3382	11150	453	15	558	5

Registry. We enable *append-only file* in Redis to log all changes for durability purposes. Moreover, we configure Redis to save snapshots every few minutes for additional reliability. To improve availability and scalability, we use 3-way replication. In our setup, Redis is deployed on all nodes of the cluster (P-servers and D-servers) so that a dedicated metadata database cluster is not needed. However, it is also possible to setup DupHunter with a dedicated metadata database cluster.

To ensure that the metadata is in a consistent state, DupHunter uses Redis' atomicity so that no file duplicates are stored in the cluster. For the file and layer indices and the slice and layer recipes, each key can be set to hold its value only if the key does not yet exist in Redis (i.e, using `SETNX` [55]). When a key already holds a value, a file duplicate or layer duplicate is identified and is removed from the registry cluster.

Additionally, DupHunter maintains a synchronization map to ensure that multiple layer restoring processes do not attempt to restore the same layer simultaneously. If a layer is currently being restored, subsequent `GET` layer requests to this layer wait until the layer is restored. Other layers, however, can be constructed in parallel.

Both the metadata database and layer store used by DupHunter are scalable and can handle large image datasets. DupHunter's metadata overhead is about 0.6% in practice, e.g., for a real-world layer dataset of 18 GB, DupHunter stores less than 100 MB of metadata in Redis.

6 Evaluation

We answer two questions in the evaluation: how do deduplication modes impact the performance–redundancy trade-off, and how effective are DupHunter's caches.

6.1 Evaluation Setup

Testbed. Our testbed consists of a 16-node cluster, where each node is equipped with 8 cores, 16 GB RAM, a 500 GB SSD, and a 10 Gbps NIC.

Dataset. We downloaded 0.93 TB of popular Docker images (i.e., images with a pull count greater than 100) with 36,000 compressed layers, totalling 2 TB after decompression. Such dataset size allowed us to quickly evaluate DupHunter's different modes without losing the generality of results. The file-level deduplication ratio of the decompressed dataset is 2.1.

Workload generation. To evaluate how DupHunter performs with production registry workloads, we use the IBM Cloud Registry traces [11] that come from four production registry clusters (`Dal`, `Fra`, `Lon`, and `Syd`) and span approximately 80 days. We use Docker registry trace player [11] to replay the first 15,000 requests from each workload as shown in Table 2. We modify the player to match requested layers in the IBM trace with real layers downloaded from Docker Hub based on the layer size[2]. Consequently, each layer request pulls or pushes a real layer. For manifest requests, we generate random well-formed, manifest files.

In addition, our workload generator uses a proxy emulator to decide the server for each request. The proxy emulator uses consistent hashing [37] to distribute layers and manifests. It maintains a ring of registry servers and calculates a destination registry server for each `push` layer or manifest request by hashing its digest. For `pull` manifest requests, the proxy emulator maintains two consistent hashing rings, one for the P-servers, and another for the D-servers. By default, the proxy first queries the P-servers but if the requested P-server is not available, it pulls from the D-servers.

Schemes. We evaluate DupHunter's deduplication ratio and performance using different deduplication and redundancy schemes. The base case considers 3-way layer replication and file-level deduplication. In that case, DupHunter provides five deduplication modes: B-mode 0, 1, 2, 3, and S-mode. Note that B-mode 0 deduplicates all layer replicas (denoted as global file-level deduplication with replication or *GF-R*) while B-mode 3 does not deduplicate any layer replicas.

To evaluate how DupHunter works with block-level deduplication, we integrate B-mode 0 with `VDO`. For each D-server, all unique files are stored on a local `VDO` device. Hence, in that mode DupHunter provides global file-level deduplication and local block-level deduplication (*GF+LB-R*).

We also evaluate DupHunter with an erasure coding policy instead of replication. We combine B-mode 0 with `Ceph` such that each D-server stores unique files on a `Ceph` erasure coding pool with global block-level deduplication enabled. We denote this scheme as *GB-EC*. We compare each scheme to Bolt [41] with 3-way replication as our baseline (*No-dedup*).

6.2 Deduplication Ratio vs. Performance

We first evaluate DupHunter's performance/deduplication ratio trade-off for all of the above described deduplication schemes. For the replication scenarios, we use 3-way replication and for GB-EC, we use a (6,2) Reed Solomon code [53, 56]. Both replication and erasure cod-

[2] The original player generates random or zeroed data for layers.

Table 3: Dedup. ratio vs. GET layer latency.

Mode	Dedup. ratio	Performance improvement (P-servers)
B-mode 1	1.5	1.6×
S-mode	1.3	2×
B-mode 2	1.2	2.6×
B-mode 3	1	2.8×
B-mode 0	Dedup ratio	Performance degradation (D-servers)
	GF-R (Global file-level [3 replicas])	
	2.1	-1.03 ×
	GF+LB-R (Global file- and local block-level [3 replicas])	
	3.0	-2.87×
	GB-EC (Global block-level [Erasure coding])	
	6.9	-6.37×

ing policies can sustain the loss of two nodes. We use 300 clients spread across 10 nodes and measure the average GET layer latency across the four production workloads. Table 3 shows the results normalized to the baseline.

We see that all four performance modes of DupHunter (B-mode 1, 2, and 3, and S-mode) have better GET layer performance compared to No-dedup. B-mode 1 and 3 reduce the GET layer latency by 1.6× and 2.8×, respectively. This is because the prefetch cache hit ratio on P-servers is 0.98 and a high cache hit ratio significantly reduces disk accesses. B-mode 3 has the highest GET layer performance but does not provide any space savings since each layer in B-mode 3 has three full replicas. B-mode 1 and 2 maintain only one and two layer replicas for each layer, respectively. Hence, B-mode 1 has a lower performance improvement (i.e., 1.6×) than B-mode 2 (i.e., 2.6×), but has a higher deduplication ratio of 1.5×. S-mode lies between B-mode 1 and 2 in terms of the deduplication ratio and performance. This is because, in S-mode, popular layers have three layer replicas while cold layers only have a single replica.

Compared to the above four modes, B-mode 0 has the highest deduplication ratio because *all* layer replicas are deduplicated. Consequently, B-mode 0 adds overhead to GET layer requests compared to the baseline performance. As shown in Table 3, if file-level deduplication and 3-way replication are used, the deduplication ratio of B-mode 0 is 2.1 while the GET layer performance is 1.03× slower.

If block-level deduplication and block-level compression are used (GF+LB-R), the deduplication ratio increases to 3.0 while the GET layer performance decreases to 2.87×. This is because of the additional overhead added by restoring the layer's files prior to restoring the actual layer. Compared to replication, erasure coding naturally reduces storage space. The deduplication ratio with erasure coding and block-level deduplication is the highest (i.e., 6.9). However, the GET layer performance decreases by 6.37× because to restore a layer, its containing files, which are split into data chunks and spread across different nodes, must first be restored.

Overall, DupHunter, even in B-mode 0, significantly decreases the layer restoring overhead compared to the naive approaches shown in Table 1 in §3.2. For example, DupHunter B-mode 0 with VDO (the GF+LB-R scheme) has a GET layer latency only 2.87× slower than the baseline compared to a the VDO-only scheme which is 60× slower compared to the baseline.

6.3 Cache Effectiveness

Next, we analyze DupHunter's caching behavior. We first study the prefetch cache and then the preconstruct cache.

6.3.1 Prefetch cache

To understand how the prefetch cache improves the P-servers' performance, we first show its hit ratio compared to two popular cache algorithms: LRU [50] and ARC [43]. Moreover, we compare DupHunter's prefetch cache with another prefetch algorithm, which makes predictions based on PUT requests [11] (denotes as ARC+P-PUT). Both of these algorithms are implemented on ARC since ARC outperforms LRU. DupHunter's prefetch algorithm, based on user behavior (UB), is denoted as ARC+P-UB. We vary the cache sizes from 5% to 15% of each workload's unique dataset size. Figure 8 shows the results for the four production workloads (Dal, Syd, Lon, and Fra).

For a cache size of 5%, the hit ratios of LRU are only 0.59, 0.58, 0.27, and 0.10, respectively. ARC hit ratios are higher compared to LRU (e.g., 1.6× Lon) because after a user pulls a layer, the user is not likely to repull this layer in the future as it is locally available. Compared to LRU, ARC maintains two lists, an LRU list and an LFU list, and adaptively balances them to increase the hit ratio.

ARC+P-PUT improves the ARC hit ratio by 1.9× for Lon. However, ARC+P-PUT only slightly improves the hit ratio for the other workloads. This is because ARC+P-PUT acts like a write cache which temporally holds recently uploaded layers and waits for the clients that have not yet pulled these layers to issue GET requests. This is not practical because the layer reuse time (i.e., interval between a PUT layer request and its subsequent GET layer request) is long. For example, the reuse time is 0.5 hr for Dal on average based on our observation. Moreover, ARC+P-PUT ignores the fact that some clients always repull layers. DupHunter's ARC+P-UB achieves the highest hit ratio. For example, ARC+P-UB's hit ratio for Dal is 0.89, resulting in a 4.2× improvement compared to ARC+P-PUT.

As shown in Figure 8, the hit ratio increases as the cache size increases. For example, when cache size increases from 5% to 15%, the hit ratio for ARC under workload Lon increases from 0.44 to 0.6. ARC+P-UB achieves the highest hit ratio of 0.96 for a cache size of 15% under workload Lon. Overall, this shows that by exploiting user behavior ARC+P-UB can achieve high hit ratios, even for smaller cache sizes.

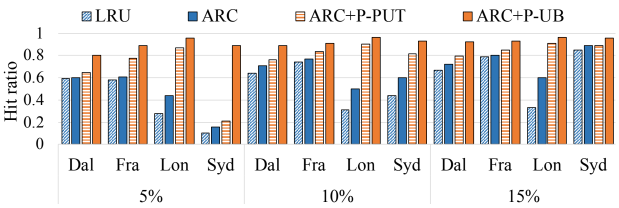


Figure 8: Cache hit ratio on P-servers with different cache algorithms.

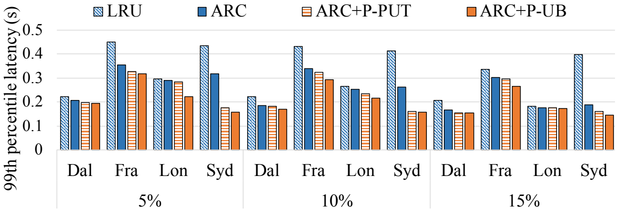


Figure 9: 99^{th} percentile GET layer latency of P-servers.

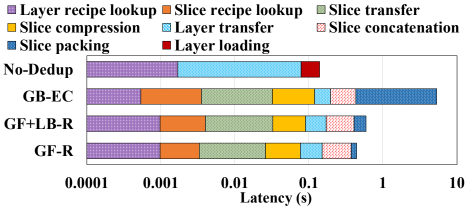


Figure 10: Layer restoring latency breakdown (X-axis is log-scale).

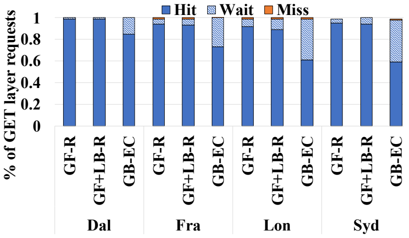


Figure 11: Preconstruct cache hit ratio.

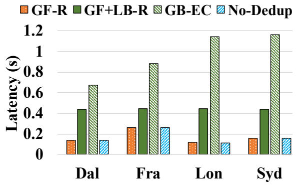


Figure 12: Performance of D-servers.

Figure 9 shows the 99^{th} percentile of GET request latencies for P-servers with different cache algorithms. The GET layer latency decreases with higher hit ratios. For example, when the cache size increases from 5% to 15%, the 99^{th} percentile latencies decrease from 0.19 s to 0.15 s for DupHunter's ARC+P-UB under workload Dal and the cache hit ratio increases from 0.8 to 0.92. Moreover, when the cache size is only 5%, ARC+P-UB significantly outperforms the other 3 caching algorithms. For example, ARC+P-UB reduces latency by 1.4 × compared to LRU for workload Fra. Overall, ARC+P-UB can largely improve GET layer performance for P-servers with a small cache size.

6.3.2 Preconstruct cache

For the preconstruct cache to be effective, layer restoring must be fast enough to complete within the time window between the GET manifest and GET layer request.

Layer restoring performance. To understand the layer restoring overhead, we disable the preconstruct cache and measure the average GET layer latency when a layer needs to be restored on D-servers. We evaluate GB-EC, GB+LB-R, and GF-R and compare it to No-dedup.

We break down the average reconstruction latency into its individual steps. The steps in layer reconstruction include looking up the layer recipe, fetching and concatenating slices, and transferring the layer. Fetching and concatenating slices in itself involves slice recipe lookup, slice packing, slice compression, and slice transfer. No-dedup contains three steps: layer metadata lookup, layer loading from disk to memory, and layer transfer.

As shown in Figure 10, GF-R has the lowest layer restoring overhead compared to GF+LB-R and GB-EC. It takes 0.44 s to rebuild a layer tarball for GF-R. Compared to the No-Dedup scheme, the GET layer latency of GF-R increases by 3.1×. Half of the GET layer latency is spent on slice concatenation. This is because slice concatenation involves writing each slice into a compressed tar archive, which is done sequentially. Slice packing and compression are faster, 0.07 s and 0.05 s, respectively, because slices are smaller and evenly distributed on different D-servers.

For the GF+LB-R scheme, it takes 0.55 s to rebuild a layer. Compared to GF-R, adding local block-level deduplication increases the overall overhead by up to 1.4× due to more expensive slice packing. It takes 0.18 s to pack a slice into an archive, 2.7× higher than GF-R's slice packing latency as reading files from the local VDO device requires an additional file restoring process.

The GB-EC scheme has the highest layer restoring overhead. The bottleneck is again slice packing which takes 5 s. This is because each file is split into four data chunks, distributed on different D-servers, and deduplicated. To pack a slice, each involved file needs to be reconstructed from different D-servers and then written to a slice archive, which incurs considerable overhead.

Preconstruct cache impact. To understand how the preconstruct cache improves D-servers' GET layer performance, we first show its hit ratio on D-servers with three deduplication schemes (GF-R, GF+LB-R, and GF-EC). The cache size is set to 10% of the unique dataset.

Figure 11 shows the preconstruct cache hit ratio breakdown. **Hit** means the requested layer is present in the cache while **Wait** means the requested layer is in the

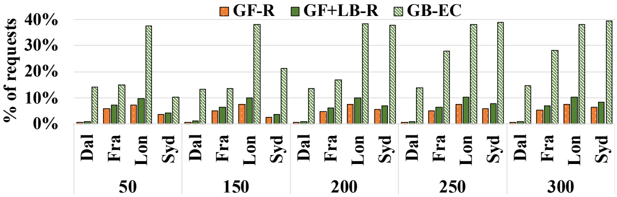


Figure 13: Request wait ratio with different number of clients.

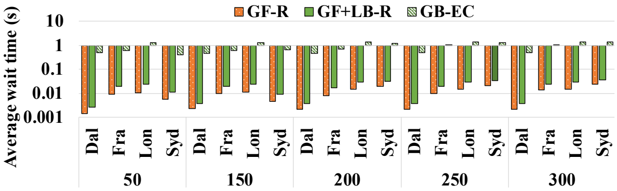


Figure 14: Average wait time with different number of clients (Y-axis is log-scale).

process of preconstruction and the request needs to wait until the construction process finishes. **Miss** means the requested layer is neither present in the cache nor in the process of preconstruction. As shown in the figure, GF-R has the highest hit ratio, e.g., 0.98 for the `Dal` workload. Correspondingly, GF-R also has the lowest wait and miss ratios because it has the lowest restoring latency and a majority of the layers can be preconstructed on time.

Note that the miss ratio of the preconstruct cache is slightly lower than that of the perfetch cache across all traces. This is because we use an in-memory buffer to hold the layer archives that are in the process of construction to avoid disk I/O. After preconstruction is done, the layers are flushed to the on-disk preconstruct cache. In this case, many requests can be served directly from the buffer and consequently, layer preconstruction does not immediately trigger cache eviction like layer prefetching. The preconstruct cache eviction is delayed til the layer preconstruction finishes.

GF+LB-R shows a slightly higher wait ratio than GF-R. Eg., the wait ratios for GF-R and GF+LB-R are 0.04 and 0.06, respectively under workload `Syd`. This is because the layer restoring latency of GF+LB-R is slightly higher than GF-R. GB-EC's wait ratio is the highest. Under workload `Syd`, 39% of `GET` layer requests are waiting for GB-EC as layers cannot be preconstructed on time.

Figure 12 shows the corresponding average `GET` layer latencies of D-servers compared to No-dedup. GF-R and GF+LB-R increase the latency by 1.04× and 3.1×, respectively, while GB-EC adds a 5× increase. This is due to GB-EC's high wait ratios.

Scalability. To analyze the scalability of the preconstruct cache under higher load, we increase the number of concurrent clients sending `GET` layer requests, and measure the request wait ratio (Figure 13) and the average wait time (Figure 14).

Under workload `Fra` and `Syd`, the wait ratio for GB-EC increases dramatically with the number of concurrent clients. For example, the wait ratio increases from 15% to 28% as the number of concurrent clients increases from 50 to 300. This is because the layer restore latency for GB-EC is higher and with more concurrent client requests, more requested layers cannot be preconstructed on time. Under workload `Lon` and `Dal`, the wait ratio for GB-EC remains stable. This is because the client requests are highly skewed. A small number of clients issue the majority of `GET` layer requests. Correspondingly, GB-EC also has the highest wait time. Under workload `Fra` and `Syd`, the average wait time increases from 0.6 s to 1.1 s and 0.4 s to 1.4 s respectively as the number of clients increases from 50 to 300 for GB-EC.

Although some layers cannot be preconstructed before the `GET` layer requests arrive, the preconstruct cache can still reduce the overhead because layer construction starts prior to the arrival of the `GET` requests. This is shown by the fact that the wait times are significantly lower than the layer construction times. For GF-R and GF+LB-R, the average wait times are only 0.001 s and 0.003 s, respectively under workload `Dal`. When the number of concurrent clients increases, the average wait time of GF-R and GF+LB-R remains low. This means that the majority of layers can be preconstructed on time for both GF-R and GF+LB-R, and the layers that cannot be preconstructed on time do not incur high overhead.

7 Conclusion

We presented DupHunter, a new Docker registry architecture that provides flexible and high performance deduplication for container images and reduces storage utilization. DupHunter supports multiple configurable deduplication modes to meet different space saving and performance requirements. Additionally, it parallelizes layer reconstruction locally and across the cluster to further mitigate overheads. Moreover, by exploiting knowledge of the application domain, DupHunter introduces a two-tier storage hierarchy with a novel layer prefetch/preconstruct cache algorithm based on user access patterns. DupHunter's prefetch cache can improve `GET` latencies by up to 2.8× while the preconstruct cache can reduce the restore overhead by up to 20.9× compared to the state of the art.

Acknowledgments

We are thankful to the anonymous reviewers and our shepherd Abhinav Duggal for their valuable feedback. This work is sponsored in part by the National Science Foundation under grants CCF-1919113, CNS-1405697, CNS-1615411, and OAC-2004751.

References

[1] Aliyun Open Storage Service (Aliyun OSS). https://cn.aliyun.com/product/oss?spm=5176.683009.2.4.Wma3SL.

[2] Docker. https://www.docker.com/.

[3] Docker Hub. https://hub.docker.com/.

[4] Dockerfile. https://docs.docker.com/engine/reference/builder/.

[5] Microsoft azure storage. https://azure.microsoft.com/en-us/services/storage/.

[6] Redis. https://redis.io/.

[7] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[8] Alfred Krohmer. Proposal: Deduplicated storage and transfer of container images. https://gist.github.com/devkid/5249ea4c88aab4c7bff1b34c955c1980.

[9] Amazon. Amazon elastic container registry. https://aws.amazon.com/ecr/.

[10] Amazon. Containers on aws. https://aws.amazon.com/containers/services/.

[11] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[12] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage. In *1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[13] Btrfs. https://btrfs.wiki.kernel.org/index.php/Deduplication.

[14] R. S. Canon and D. Jacobsen. Shifter: Containers for HPC. In *Cray User Group*, 2016.

[15] Z. Cao, H. Wen, F. Wu, and D. H. Du. {ALACC}: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 309–324, 2018.

[16] Ceph. https://docs.ceph.com/docs/master/dev/deduplication/.

[17] Cloud Native Computing Foundation Projects. https://www.cncf.io/projects/.

[18] B. Compression and Deduplication. https://tinyurl.com/vgvb7wu.

[19] Datadog. 8 Surprising Facts about Real Docker Adoption. https://www.datadoghq.com/docker-adoption/.

[20] Docker. Docker Registry. https://github.com/docker/distribution.

[21] Docker. Docker Registry HTTP API V2. https://github.com/docker/distribution/blob/master/docs/spec/api.md.

[22] DockerSlim. https://dockersl.im.

[23] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *USENIX Annual Technical Conference (ATC)*, 2014.

[24] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[25] Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu. AA-Dedupe: An Application-aware Source Deduplication Approach for Cloud Backup Services in the Personal Computing Environment. In *IEEE International Conference on Cluster Computing (Cluster)*, 2011.

[26] GNU Tar. Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html.

[27] Google. Google container registry. https://cloud.google.com/container-registry/.

[28] Google compute engine. Google Compute Engine. https://cloud.google.com/compute/.

[29] K. Gschwind, C. Adam, S. Duri, S. Nadgowda, and M. Vukovic. Optimizing Service Delivery with Minimal Runtimes. In *International Conference on Service-Oriented Computing (ICSOC)*, 2017.

[30] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[31] IBM Cloud Kubernetes Service. Ibm cloud kubernetes service. https://www.ibm.com/cloud/container-service.

[32] IBM Cloud Kubernetes Service. S3 storage driver. https://docs.docker.com/registry/storage-drivers/s3/.

[33] K. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Middleware Industry Track Workshop*, 2011.

[34] jdupes. https://github.com/jbruchon/jdupes.

[35] JFrog Artifcatory. https://jfrog.com/artifactory/.

[36] K. Jin and E. L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *International Systems and Storage Conference (SYSTOR)*, 2009.

[37] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997.

[38] K. Kumar and M. Kurhekar. Economically Efficient Virtualization over Cloud Using Docker Containers. In *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2016.

[39] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that use Inline Chunk-based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[40] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[41] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a Scalable Docker Registry via Hyperconvergence. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2019.

[42] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *International Systems and Storage Conference (SYSTOR)*, 2012.

[43] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[44] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[45] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[46] Microsoft. Azure container registry. https://azure.microsoft.com/en-us/services/container-registry/.

[47] Microsoft Azure. https://azure.microsoft.com/en-us/.

[48] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *ACM SIGOPS Operating Systems Review*, volume 35, 2001.

[49] M. Oh, S. Park, J. Yoon, S. Kim, K. Lee, S. Weil, H. Y. Yeom, and M. Jung. Design of global data deduplication for a scale-out distributed storage system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, 2018.

[50] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[51] OpenStack Swift storage driver. Openstack swift storage driver. https://docs.docker.com/registry/storage-drivers/swift/.

[52] J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[53] J. S. Plank, M. Blaum, and J. L. Hafner. Sd codes: erasure codes designed for how storage systems really fail. In *FAST*, pages 95–104, 2013.

[54] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically Debloating Containers. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[55] Redis. SETNX. https://redis.io/commands/setnx.

[56] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[57] P. Shilane, R. Chitloor, and U. K. Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[58] H. Shim, P. Shilane, and W. Hsu. Characterization of Incremental Data Changes for Efficient Data Protection. In *USENIX Annual Technical Conference (ATC)*, 2013.

[59] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo. Carving Perfect Layers out of Docker Images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[60] R. P. Spillane, W. Wang, L. Lu, M. Austruy, R. Rivera, and C. Karamanolis. Exo-clones: Better Container Runtime Image Management Across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[61] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[62] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. A Long-Term User-Centric Analysis of Deduplication Patterns. In *32nd International Conference on Massive Storage Systems and Technology (MSST)*, 2016.

[63] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *Ottawa Linux Symposium*, 2014.

[64] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017.

[65] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (ATC)*, 2018.

[66] A. Upadhyay, P. R. Balihalli, S. Ivaturi, and S. Rao. Deduplication and compression techniques in cloud design. In *2012 IEEE International Systems Conference SysCon 2012*, pages 1–6. IEEE, 2012.

[67] Vdo. https://github.com/dm-vdo/vdo.

[68] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[69] E. Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[70] ZFS. https://en.wikipedia.org/wiki/ZFS.

[71] F. Zhao, K. Xu, and R. Shain. Improving Copy-on-Write Performance in Container Storage Drivers. In *Storage Developer Conference (SDC)*, 2016.

[72] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the docker hub dataset. In *IEEE International Conference on Cluster Computing (Cluster)*, 2019.

[73] R. Zhou, M. Liu, and T. Li. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[74] B. Zhu, K. Li, and R. H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems

Yu Zhang[†], Ping Huang[†§], Ke Zhou[†*], Hua Wang[†], Jianying Hu[‡], Yongguang Ji[‡], Bin Cheng[‡]
[†]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology,*
Intelligent Cloud Storage Joint Research center of HUST and Tencent
[§]*Temple University,*[‡]*Tencent Technology (Shenzhen) Co., Ltd.*
[*]*Corresponding author: zhke@hust.edu.cn*
○ *Yu Zhang and Ping Huang are the co-first authors*

Abstract

We propose an Online-Model based Scheme for Cache Allocation for shared cache servers among cloud block storage devices. *OSCA* can find a near-optimal configuration scheme at very low complexity improving the overall efficiency of the cache server. *OSCA* employs three techniques. First, it deploys a novel cache model to obtain a miss ratio curve (MRC) for each storage node in the cloud infrastructure block storage system. Our model uses a low overhead method to obtain data reuse distances from the ratio of re-access traffic to the total traffic within a time window. It then translates the obtained reuse distance distribution into miss ratio curves. Second, knowing the cache requirements of storage nodes, it defines the total hit traffic metric as the optimization target. Third, it searches for a near optimal configuration using a dynamic programming method and performs cache reassignment based on the solution. Experimental results with real-world workloads show that our model achieves a Mean Absolute Error (MAE) comparable to existing state-of-the-art techniques, but we can do without the overheads of trace collection and processing. Due to the improvement of hit ratio, *OSCA* reduces IO traffic to the back-end storage server by 13.2% relative to an equal-allocation-to-all-instances policy with the same amount of cache memory.

1 Introduction

With widespread deployment of the cloud computing paradigm, the number of cloud tenants have significantly increased during the past years. To satisfy the rigorous performance and availability requirements of different tenants, cloud block storage (CBS) systems have been widely deployed by cloud providers (e.g., AWS, Google Cloud, Dropbox, Tencent, etc.). As revealed in previous studies [4, 13, 18, 40], cloud infrastructures typically employ cache servers, consisting of multiple cache instances competing for the same pool of resources. Judiciously designed cache policies play an important role in ensuring the stated service level objectives (SLO).

The currently used even-allocation policy called EAP or equal cache partitioning [41] determines the cache requirements in advance according to the respective subscribed SLOs and then provisions cache resources for each cache instance. However, this static configuration method is often suboptimal for the cloud environment and induces resource wastage, because the cloud I/O workloads are commonly highly-skewed [3, 16, 20].

In this paper, we aim to address the management of cache resources shared by multiple instances of a cloud block storage system. We propose an Online-Model Scheme for dynamic Cache Allocation (OSCA) with miss ration curves (MRC). *OSCA* does not require to separately obtain traces to construct MRCs. *OSCA* searches for a near-optimal configuration scheme at a very low complexity and thus improves the overall effectiveness of cache service. Specifically, the core idea of *OSCA* is three-fold. First, *OSCA* develops an online cache model based on re-access ratio (Section 3.2) to obtain the cache requirements of different storage nodes with low complexity. Second, *OSCA* uses the total hit traffic as the metric to gauge cache efficiency as the optimization target. Third, *OSCA* searches for an optimal configuration using dynamic programming method. Our approach is complementary to the most recent on-line scheme SHARDS [34]. It can achieve a suitable trade-off between computation complexity and space overhead (Section 2.3).

As the key contribution, we propose a Re-Access Ratio based Cache Model (*RAR-CM*) to construct the MRC and calculate the space requirements of each cache instance. Compared with previous models, *RAR-CM* does not need to collect and process traces, which can be expensive in many scenarios. Instead, we shift the cost of processing I/O traces to that of tracking the unique data blocks in a workload (i.e., the working set), and this proves advantageous when the number of unique blocks can be efficiently processed in memory. We experimentally demonstrate the efficacy of *OSCA* using an in-house CBS simulator with I/O traces collected from a CBS production system. We are in the process of releasing those traces to the SNIA IOTTA repository [27].

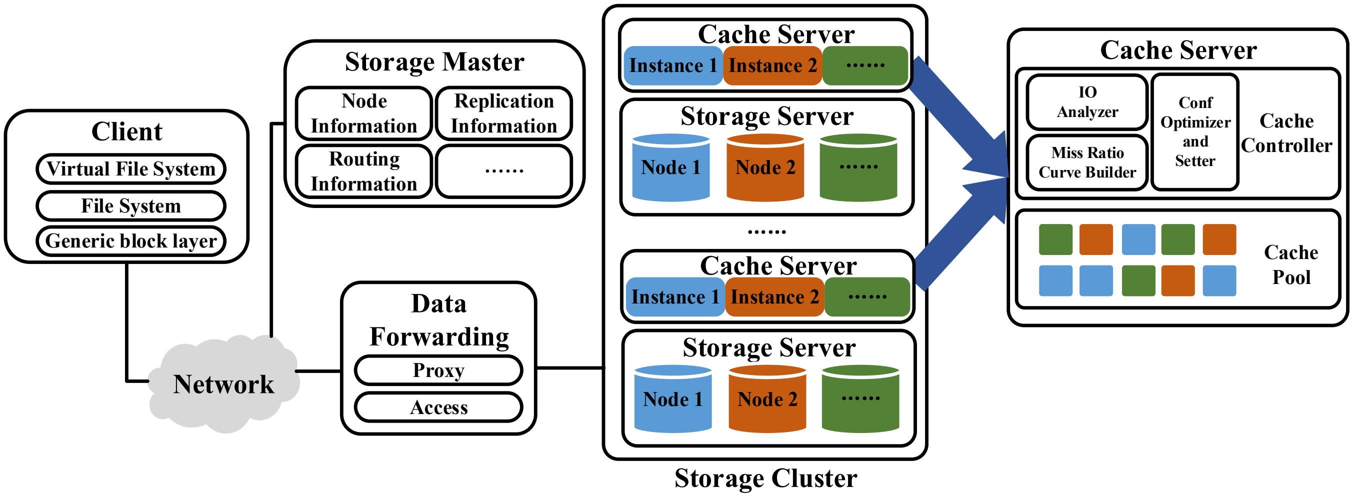


Figure 1: The architectural view of a cloud block storage system (CBS), which includes a client cloud disk layer, Data Forwarding layer, and Storage Cluster containing multiple storage servers each of which is paired with a cache server. The cache server is divided into multiple cache instances respectively responsible for the nodes (i.e., disks) in the corresponding storage server.

The rest of this paper is structured as follows. In Section 2, we introduce the background and motivation of this study and take a detailed look at existing cache modeling methods. In Section 3, we elaborate on the details of our *OSCA* cache management policy. In Section 4, we present our experimental method and the results. In Section 5, we discuss the related work and conclude in Section 6.

2 Background and Motivation

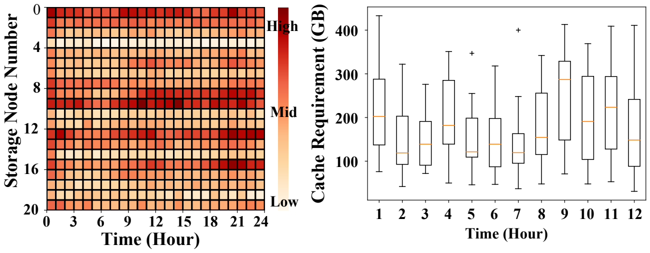
2.1 Cloud Block Storage

To provide tenants with a general, reliable, elastic and scalable block-level storage service, cloud block storage (CBS) has been developed and deployed extensively by the majority of cloud providers. CBS is made up of client layer, data forwarding layer, and storage server layer. The client layer presents tenants with the view of elastic and isolated logic cloud disks allocated according to the tenants' configuration and mounted to the client virtual machines. The data forwarding layer maps and forwards I/O requests from the client-end to the storage server-end. The storage server layer is responsible for providing physical data storage space and it typically employs replication to ensure data reliability and availability. More specifically, a CBS contains multiple components, the client, the storage master, the proxy and access server, and the storage server (as shown in Fig. 1). These components are interconnected through fast fiber-optic networks. The client provides the function of cloud disk virtualization and presents the view of cloud disks to tenants. The storage master (also called the metadata server) assumes the management of node information, replication information, and data routing information. The proxy server is responsible for external and internal storage protocol conversion. In our work, the I/O trace collection tasks are conducted on the proxy server. The access server is responsible for I/O routing that determines which storage node should an access be assigned to based on the MD5 digest calculated from the information of the record. It uses consistent hashing to map each MD5 digest to a positive integer denoting storage node. The storage server consists of multiple failure domains to reduce the probability of correlated failures. Storage servers allocate physical space from conventional hard disk drives, whose performance alone often cannot meet the requirements of cloud applications dominated by random accesses. Therefore, a CBS system typically employs a cache server (comprised of SSDs [18], NVMs [11], or other emerging storage technologies [20]) to improve performance.

As indicated in Fig. 1, the cache server includes a cache controller and a cache pool. To ensure scalability, there are often multiple cache instances, each associated with one storage node, at the cache server. The user-perceived cloud disk is a collection of logical blocks commonly spread across several physical node disks. A single physical disk is thus shared by multiple virtual disks. As a result, the accesses to a physical disk are mixed patterns. A cache instance is deployed to perform caching for each physical disk and our task is to partition the cache resource among all the cache instances.

2.2 Cache Allocation Scheme

The cache allocation scheme, which is responsible for cache resource assignment, largely influences the efficiency of the cache server. Even-allocation policy (EAP), where each block storage instance receives the same pre-determined amount of cache, is typically used in real production systems for its simplicity. The EAP first analyzes the total cache space requirements in advance according to the defined service-level objectives, and then uniformly allocates cache resources for each cache instance. In essence, it is a static allocation policy and suffers from cache underutilization if over-provisioned and performance degradation if under-provisioned, especially in the cloud environment featuring highly-skewed workloads with unpredictable and irregular dynamics [3, 16, 20]. As shown in Fig. 2 (a), we randomly selected 20 storage nodes and present their IO traffic lasting a period of 24 hours. The figure confirms that the traffic is unevenly distributed to the storage nodes in the realistic CBS production system. Presented from a different perspective, Fig. 2 (b) shows the distribution of cache requirements of those 20 storage nodes during the first 12 hours in order to reach for a level of 95% hit ratio. Again, it shows each storage node has different cache requirements at different times.



(a) (b)

Figure 2: Fig. (a) presents the frequency of accesses over storage nodes in a typical 24 hour period observed in our traces. The color indicates the intensity of accesses, measured by requests per seconds arriving at each storage node in one-hour time window. The darker the red color in the figure, the more intensive the I/O traffic is. Fig. (b) shows the distribution of cache requirements of those 20 storage nodes during the first 12 hours in order to reach for a level of 95% hit ratio. The orange horizontal line in each box denotes the median cache requirement of the 20 storage nodes, while the bottom and top side of the box represent the quartiles and the lines that extend out of the box (whiskers) represent data outside the upper and lower quartiles.

To improve this policy via ensuring more appropriate cache allocations, there have been proposed two broad categories of solutions. The first category is intuition-based policies such as TCM [19], REF [42], which are **qualitative methods** based on intuition or experience. These policies often provide a feasible solution to the combined optimization problem at an acceptable computation and space cost. For example, according to memory access characteristics, TCM categorizes threads as either latency-sensitive or bandwidth-sensitive and correspondingly prioritizes the latency-sensitive threads over the bandwidth-sensitive threads as far as cache allocation concerns. Such coarse grained qualitative methods are heavily dependent on prior reliable experiences or workload regularities. Therefore, their efficacy is not guaranteed for cloud workloads which are diverse and constantly changing.

The other category is model-based policies, which are **quantitative methods** enabled by cache models typically described by Miss Rate Curves (MRCs), which plot the ratio of cache misses to total references, as a function of cache size [14, 29, 33, 34]. Compared with intuition-based policies, model-based policies are based on cache models containing information about dynamic space requirements of each cache instance and thus are to result in a near-optimal solution. The biggest challenge with quantitative methods lies in constructing accurate miss rate curves at practically acceptable computational and space complexity in an online manner. Most cache models rely on offline analysis due to the enormous computation complexity and space overhead, limiting their practical applicability. A host of research efforts have been conducted to cost-effectively construct miss rate curves with the goal to enable realistic online MRC profiling [4, 29, 31, 33, 34]. Especially, the most recent proposed Spatially Hashed Approximate Reuse Distance Sampling (SHARDS) [34] is an on-line cache model which takes constant space overhead and significantly reduced computational complexity, yet still generating highly accurate MRCs. (Section 2.3 presents more details about SHARDS).

2.3 Existing Cache Modeling Methods

The biggest obstacle to apply an optimal policy to a real system is the huge computational complexity and storage overhead involved to construct accurate cache models which are used to obtain the space requirement of each cache instance. Existing commonly-used cache modeling methods can be divided into two categories, the cache modeling based on locality quantization method and simulation method.

Locality quantization method analyzes the locality characteristics (e.g., Footprint [39], Reuse Distance [34], Average Eviction Time [14], etc.) of workloads and then translates these characteristics into miss ratio curves [7]. The miss ratio curve indicates the miss ratio corresponding to different cache sizes, which can be leveraged to quantitatively determine the cache requirements of different storage nodes. The most commonly used locality characteristic is the Reuse Distance Distribution (as shown in Fig. 3). The reuse distance is the amount of unique data blocks between two consecutive accesses to the same data block. For example, suppose a reference sequence is A-B-C-D-B-D-A, the reuse distance of data block A is 3 because the unique data set between two successive accesses to A is {B, C, D}. The reuse distance is workload-specific and its distribution might change over time.

The distribution of reuse distance has a great influence on the cache hit ratio. More specifically, a data block hits the cache only when its reuse distance is smaller than its eviction distance which is defined as the amount of unique

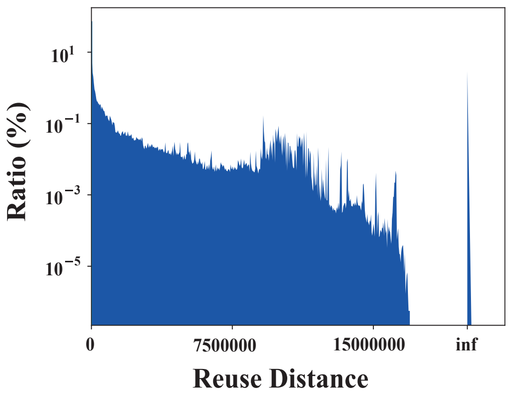


Figure 3: Reuse distance distribution of a one-day long trace from a CBS storage node.

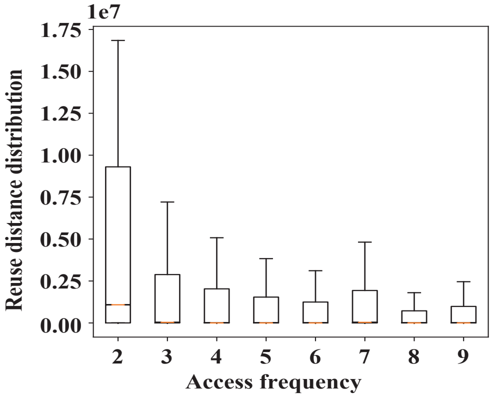


Figure 4: The reuse distance distribution of blocks of a one-day long trace from a CBS storage node, grouped by the access frequencies.

blocks accessed from the time it enters the cache to the time it is evicted from the cache. For a given sequence of block reference, the eviction distance of each block is dependent on the adopted cache algorithm. Different cache algorithms could lead to different eviction distances even for the same block in the reference sequence. The LRU algorithm uses one list and always puts the most recently used data block at the head of the list and only evicts the least recently used block at the tail of the list. As a result, the eviction distance of the most recently used block is equal to the cache size. 2Q [26], ARC [23], and LIRS [17] use two-level LRU lists and a data block can enter the second level lists only when it has been hit in the first level list before. Therefore, these algorithms can result in larger eviction distance for the blocks which have been accessed twice. Similarly, MQ [45] uses multiple-level LRU lists and it causes data blocks with more access frequencies to have larger eviction distances.

In this paper, we focus on modeling LRU algorithm for two reasons. First, LRU is widely deployed in many real cloud caching systems [15, 21]. Second, based on our analysis results of realistic cloud cache, when the cache size becomes larger than a certain size, the advanced algorithms would degenerate to LRU. Fig. 4 presents the reuse distance distribution of blocks with different access frequencies using a one-day long trace from a CBS storage node. The trace is collected from Tencent CBS [30] and we are in the process of making it publicly available via the SNIA IOTTA repository [27]. The bottom and top of each box represent the minimum and maximum reuse distance. The reuse distances of blocks whose access frequencies are larger than 2 are smaller than 0.75×10^7. Therefore, when the cache size becomes larger than 229 GB (0.75×10^7 blocks, each size being 32 KB), the data blocks whose frequencies are larger than 2 can all be hit in the LRU cache because their reuse distances are smaller than the cache size. Other advanced algorithms (e.g., 2Q , ARC, and LIRS) which cause blocks whose occurrences are larger than 2 to have larger eviction distance would degenerate to LRU [44]. Therefore, in our caching system where cache size for each storage node is close to 229 GB (assuming EAP is deployed), the performance differences between LRU and other algorithms are negligible.

Existing cache modeling methods (ours included) calculate the hit ratio of the LRU algorithm as the discrete integral sum of the reuse distance distribution (from zero to the cache size) curve (as shown in Eq. 1).

$$hr(C) = \sum_{x=0}^{C} rdd(x) \tag{1}$$

In the above equation, *hr(C)* is the hit ratio at cache size C and *rdd(x)* denotes the distribution function of reuse distance. However, obtaining the reuse distance distribution has an $O(N * M)$ complexity, where N is the total number of references in the access sequence and M is number of the unique data blocks of references [22]. Recent studies have proposed various ways to decrease the computation complexity to $O(N * log(n))$ using Search Tree [24], Scale Tree [43], Interval Tree [1]. These methods use a balanced tree structure to get a logarithmic search time upon each reference to calculate block reuse distances.

SHARDS [34], further decreases the computation complexity with fixed amount of space. To build MRCs, SHARDS first selects a representative subset of the traces through hashing block addresses. It then inputs the selected traces to a conventional cache model to produce MRCs. Since SHARDS only needs to process a subset of the traces, it significantly reduces the computation overheads and memory space to host the traces. Therefore, SHARDS has the potential to be applied in an on-line manner. All sampled traces can be stored in a given amount of memory by dynamically adjusting the sample ratio. It should be noted that it requires to rescale up the results to obtain the eventual reuse distance for the original traces.

In this paper, we propose an on-line cache model called

RAR-CM to build MRC which is based on a metric called re-access ratio. Our approach does not rely on collecting traces beforehand. Both our approach and SHARDS can be practically applied on-line. Our approach is different from SHARDS in the following aspects. First, SHARDS uses a sampled subset of traces to construct MRCs, while our approach processes I/O requests inline and does not store or process a separate I/O trace. Second, on average it takes $O(lg(M*R))$ asymptotic complexity for SHARDS to update the information in the balanced tree for every sampled block access, where M is the total number of unique blocks in the trace. Our approach only requires to update two counters and thus is $O(1)$.

Table 1 summarizes the comparison between SHARDS and *RAR-CM* in four primary aspects. M, n, and R denotes the total number of unique blocks, the maximum number of records that can be contained in the fixed memory(SHARDS), and the sampling ratio (SHARDS). From the table, we can see that both SHARDS and *RAR-CM* can potentially be applied to construct MRCs in an on-line manner. We can choose to use either of them based on specific scenarios. A general guidance is if we are more concerned about saving computational resources and the available memory can hold support all unique blocks, then our *RAR-CM* is the choice. If we are more constrained by memory and computing resources is not an issue (e.g., we have GPU available), then SHARDS is the choice. In fact, SHARDS and *RAR-CM* are two similar and complementary approaches that can achieve an optimal trade-off point between computation complexity and space overhead. As can be seen from Table 1, one major disadvantage with our approach is that it requires $O(M)$ space to store the information about each unique block. Therefore, in cases where memory is constrained and the working set is relatively large, SHARDS is a better choice.

Table 1: The comparison of *RAR-CM* and SHARDS. M is the number of unique data blocks in the access stream. R denotes the sampling ratio in SHARDS, and n is the number of the sampled unique blocks in the fixed memory. Reuse distribution generation complexity is O(1) for both methods.

	SHARDS	*RAR-CM*
Use full trace	No	Yes
Space Complexity	$O(M*R)$ fixed sample $O(1)$ fixed memory	$O(M)$
Block Access Overhead	$O(log(M*R))$ fixed sample $O(log(n))$ fixed memory	$O(1)$

Simulation-based cache modeling and recently proposed miniature simulation based on the idea of SHARDS [33] need to concurrently run multiple simulation instances to determine the cache hit ratio in different cache sizes. While SHARDS can be applied on-line to process currently sampled traces to obtain the miss ratio curve, the miniature simulation constructs the miss ratio curves based on collected trace beforehand, which could incur no-trivial overhead. We have conducted an experiment with the miniature simulation [33]. Specifically, we run 20 simulation routines (each routine starts 20 threads) simultaneously on a 12-core CPU (i.e., Intel Xeon CPU E5-2670 v3), and this method takes around 69 minutes to analyze a one-day-long IO trace file and most of the time is consumed in trace reading (1.067 µs / record) and IO mapping (2.406 µs / record).

3 Design and Implementation

3.1 Design Overview

OSCA performs three steps, online cache modeling, optimization target defining, and the optimal configuration searching. Fig. 5 illustrates the overall architecture of *OSCA*. Upon receiving a read request from the client, CBS first partitions and routes the request to the storage node and finds the data in the index map of the corresponding cache instance. If it is found in the map on the cache server, the data will be returned to the client directly, and the request will not need to go to the storage server node. Otherwise, the data located in the corresponding physical disk is fetched and returned. A write request is always first written to the cache, and then flushed to the back-end HDD storage asynchronously. All I/O requests are monitored and analyzed by the cache controller for cache modeling. Then the cache controller will find the optimal configuration scheme according to the cache model and the optimization target and finally reassign the cache resource for each cache instance periodically.

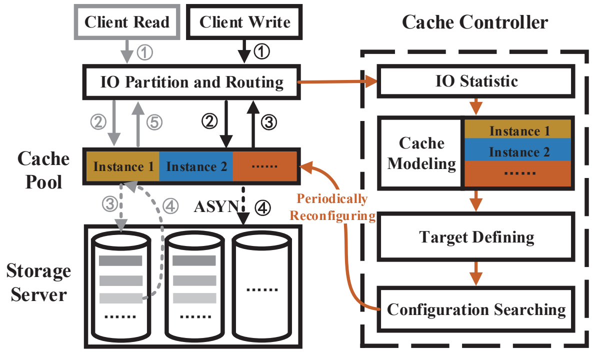


Figure 5: The overall architecture of *OSCA*. Each cache instance is paired with a physical disk which provides storage space for cloud disks. The cache controller monitors the access traffic to physical disks and construct cache models to guide the reassignment of cache resources among cache instances.

3.2 Re-access Ratio Based Cache Model

The main purpose of cache modeling is to obtain the miss ratio curve, which describes the relationship between miss ratio and cache size. The resultant curve can be used in practical applications to instruct cache configurations. We propose a novel online re-access ratio cache model (*RAR-CM*), which can be constructed without the computational overhead of trace collection and processing, when compared with existing cache models. Fig. 6 shows the main components of *RAR-CM*. For a request to block B, we first check its history information in a hash map and obtain its last access timestamp (*lt*) and last access counter (*lc*, a 64-bit number denoting the total number of requests which have been seen so far at the time of last access timestamp, or equivalently the block sequence number of the last reference to block B). We then use *lt*, *lc* and *RAR* curve to calculate the reuse distance of block B. Then the resultant reuse distance is used to calculate the miss ratio curve.

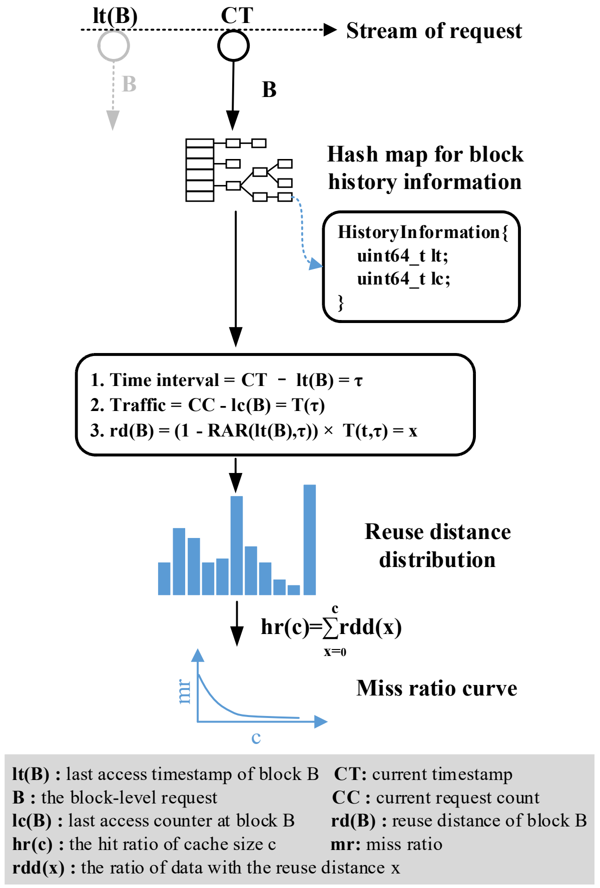

lt(B)
CT
Stream of request
B
B
Hash map for block history information
HistoryInformation{
uint64_t lt;
uint64_t lc;
}
1. Time interval = CT - lt(B) = τ
2. Traffic = CC - lc(B) = T(τ)
3. rd(B) = (1 - RAR(lt(B),τ)) × T(t,τ) = x
Reuse distance distribution
hr(c)=∑rdd(x) c x=0
Miss ratio curve
mr
c
lt(B) : last access timestamp of block B
CT: current timestamp
B : the block-level request
CC : current request count
lc(B) : last access counter at block B
rd(B) : reuse distance of block B
hr(c) : the hit ratio of cache size c
mr: miss ratio
rdd(x) : the ratio of data with the reuse distance x

Figure 6: The overview of re-access ratio based cache modeling. It calculates the reuse distance using re-access ratio and then constructs the miss rate curve based on reuse distance.

RAR, which is defined as the ratio of the re-access traffic to the total traffic during a time interval τ after time t, is expressed as $RAR(t,\tau)$. It essentially represents a metric reflecting how blocks in the following time interval are re-accessed. Fig. 7 shows the re-access ratio during a time interval τ with block access sequence {A, B, C, D, B, D, E, F, B, A}. The number of reaccessed blocks (which includes reaccess to the same block, e.g., B) is 4 (the blue letters marked in Fig. 7), and the total traffic is 10. Therefore, we obtain $RAR(t,\tau)$ = 4 / 10 = 40%.

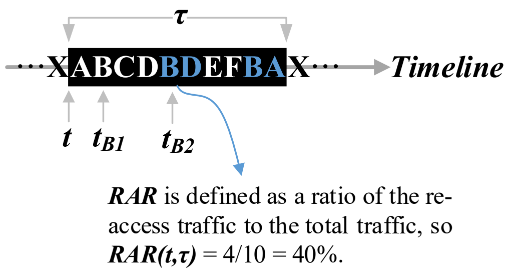

τ
···XABCDBDEFBAX···
Timeline
t
tB1
tB2
RAR is defined as a ratio of the re-access traffic to the total traffic, so RAR(t,τ) = 4/10 = 40%.

Figure 7: The definition of re-access ratio of an access sequence during a time period $[t, t+\tau]$.

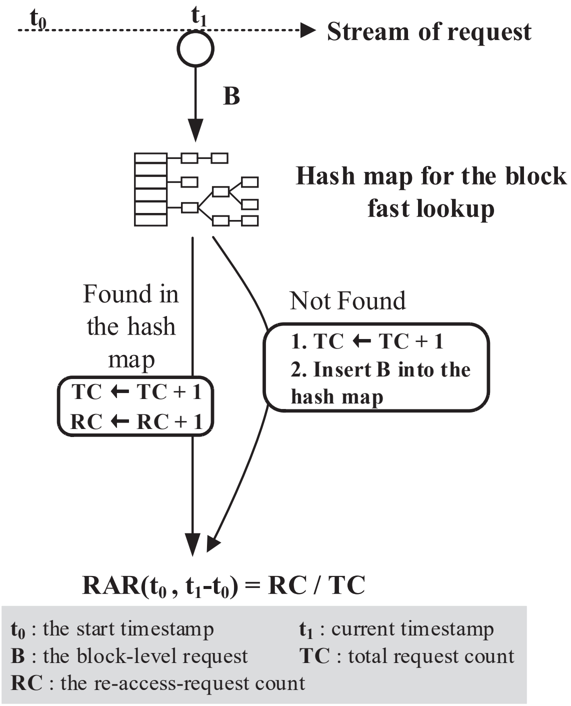
We use the obtained *RAR* for cache modeling because it has a number of favorable properties:

- It can be easily translated to the locality characteristics.
- It can be obtained with low overhead given it's complexity of O(1).
- It can be stored with low overhead of memory footprint.

Locality characteristics. *RAR* can be translated to the commonly used footprint and reuse distance characteristics. As mentioned, the reuse distance is the unique accesses between two consecutive references to the same data block. Assuming that the time interval between two consecutive references of block B is τ, then the reuse distance of block B, $rd(B)$, can be represented by Eq. 2, where $RAR(t,\tau)$ and $T(t,\tau)$ means the re-access ratio and total block accesses between the two consecutive references to block B, respectively. t indicates the last access timestamp of block B. For instance, to calculate the reuse distance of the second B at time t_{B2}, we use $t_{B2} - t_{B1}$ as the τ value for RAR function and 3 as the value of $T(t,\tau)$ in Eq. 2.

$$rd(B) = (1 - RAR(t,\tau)) \times T(t,\tau) \tag{2}$$

Complexity of O(1). Fig. 8 describes the process of obtaining the re-access ratio curve. $RAR(t_0,t_1\text{-}t_0)$ is calculated by dividing the re-access-request count (RC) by the total request count (TC) during $[t_0,t_1]$. To update RC and TC, we first lookup the block request in a hash map to determine whether it is a re-access-request. If found, it is a re-access-request and both TC and RC should be increased by 1. Otherwise, only TC is increased by 1.



t_0
t_1
Stream of request
B
Hash map for the block fast lookup
Found in the hash map
TC ← TC + 1
RC ← RC + 1
Not Found
1. TC ← TC + 1
2. Insert B into the hash map
RAR(t_0 , t_1-t_0) = RC / TC
t_0 : the start timestamp
t_1 : current timestamp
B : the block-level request
TC : total request count
RC : the re-access-request count

Figure 8: The process of obtaining re-access ratio curve. For each incoming block access, it only needs to update two counters, i.e., RC and TC.

Memory footprint. Fig. 9 shows the RAR curves calculated at the end of each of the six trace days. As can be seen, those curves have similar shapes and can be approximated by logarithmic curves which have the form of $RAR(\tau) = a * log(\tau) + b$, where τ is the time variable. Therefore, we only store the two parameters to represent the curve, which has negligible overhead. Note that the presented logarithmic curves are obtained from our traces. Others ways of compactly representing the distribution are possible (e.g., a Weibull distribution [36]). Moreover, for different workloads the shapes of the *RAR* curves may vary and correspondingly we could approach that with other distributions.

In summary, we calculate the *RAR* curve using a hash map to decide whether a block reference is a re-access or not and then based on the *RAR* curve we obtain the reuse distance distribution according to Eq. 2. Finally, the reuse distance distribution is translated to the miss ratio curve leveraging Eq. 1. With the miss ratio curve in place, we then perform cache reconfiguration. Ideally, we want to obtain all the *RAR* curve at each timestamp which is cost-ineffective. Fortunately, we observe that $RAR(t, \tau)$ is relatively insensitive to time t by analyzing a week-long cloud block storage trace (a mixed-trace consisting of tens of thousands of cloud disks' requests). Specifically, although cloud workloads are highly dynamic, we observe that the RAR curves are stable over a couple of days, which means changes of *RAR* curve are negligible over days. Therefore, in our experiment we only calculate the *RAR* curve once a day to represent the *RAR* curve for the next coming day. Specifically, assume the starting time of next day is t_0 and a block is accessed at time t_1. Then we use $t_1 - t_0$ as input to the *RAR* curve function to calculate it's reuse distance using Eq. 2. Note that if the block is accessed the first time, then it's reuse distance is to set to infinitely large, meaning it is a miss.

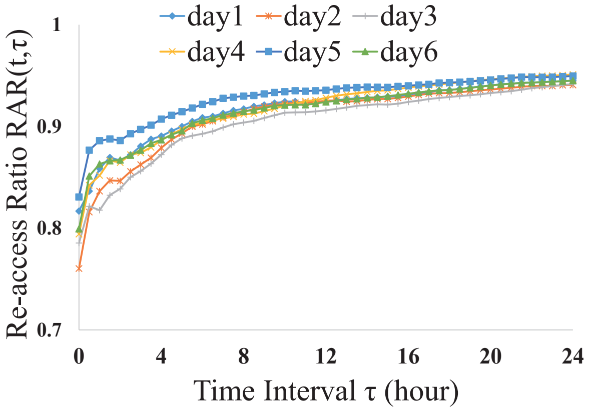


Figure 9: The *RAR* curves of the six days are similar and can be fitting-curved using as logarithmic functions. These *RAR* curves are calculated based on the traces collected from one storage node of Tencent CBS.

3.3 Optimization Target

After obtaining cache modeling, we should define a cache efficiency function as the optimization target. Previous studies have suggested a number of different optimization target (e.g. RECU [41], REF [42], et al.). For instance, RECU considers the elastic miss ratio baseline (EMB) and the elastic space baseline (ECB) to balance tenant-level fairness and the overall performance. Considering our case being cloud server-end caches, in this work we use the function E in Eq. 3 as our optimization target. $HitRatio_{node}$ represents the hit rate of the node and $Traffic_{node}$ denotes the I/O traffic to this node. Therefore, this expression represents the overall hit traffic among all nodes. The bigger the value of E is, the less traffic is sent to the backend HDD storage. Admittedly, other optimization targets are also possible and can be decided taking into service level objective account. Based on this target function, our aim is to find a cache assignment method which leads to the largest hit traffic and the smallest traffic to the back-end storage server.

$$E = \sum_{node=1}^{N} HitRatio_{node} \times Traffic_{node} \qquad (3)$$

3.4 Searching for Optimal Configuration

Based on the cache modeling and defined target mentioned above, our *OSCA* searches for the optimal configuration scheme. More specifically, the configuration searching process tries to find the optimal combination of cache sizes of each cache instance to get the highest efficiency E.

To speed up the search process, we use dynamic programming (DP), since a large part of calculations are repetitive. A DP method can avoid repeated calculations using a table to store intermediate results and thus reduce the exponential computational complexity to a linear level.

3.5 Implementation Details

Algorithm 1 presents the pseudocode of the process of our *RAR-CM*. The content of block history information is shown in Fig. 6. The re-access ratio curve and the reuse distance distribution are arrays. The subroutine *update_reuse_distance* (Algorithm 2) is used to update the reuse distance distribution *RD* according to the re-access ratio curve *RAR*. And the subroutine *get_miss_ratio_curve* (Algorithm 3) is used to obtain the miss ratio curve according to the reuse distance distribution *RD*. Specifically, *RD* is formed by an array containing 1024 elements, each denoting 1 GB wide (32768 cache blocks of size 32 KB), representing the reuse distances up to 1 TB. The *get_miss_ratio_curve* calculates the cumulative distribution function for *RD*.

From the pseudocode, we can know that the reuse distance calculation of each block is very lightweight which only involves several simple operations and takes hundreds of nanoseconds. This means *RAR-CM* has a negligible influence on the storage server. And the history information of each referenced block contains two 64-bit numbers, occupying very little memory space. More details for the discussion of CPU, memory, network usage can be referenced to Section 4.5.

4 Evaluation

4.1 Experimental Setup

Trace Collection. To evaluate *OSCA*, we have collected six-day long I/O traces from a production cloud block storage system using a proxy server which is responsible for I/O forwarding between client and storage server. The cloud block storage system has served tens of thousands of cloud disks. The trace files record every I/O request issued by the tenants and each item of the trace file contains the *request timestamp*, *cloud disk id*, *request offset*, *I/O size*, and so on. To not influence tenants' I/O performance, we have optimized the collection tasks by merging and reporting I/O traces to the trace storage server periodically. We trigger the collection tasks to scan the local I/O logs on the proxy server and report the merged I/O traces every hour, which is an appropriate time interval that can balance the number of tasks with the size of the merged trace files.

Algorithm 1: The pseudocode of the *RAR-CM* process

```
Data: Initialize the global variable: hash map for block
      history information $H$, current timestamp $CT$,
      current block sequence number $CC$, and the
      re-reference count $RC$. The re-access ratio curve
      $RAR$. The reuse distance distribution $RD$
Input: a sequence of block accesses
Output: output the miss ratio curve
while has unprocessed block access do
    $B \leftarrow next\ block$
    $CC \leftarrow CC + 1$
    $CT \leftarrow current\ timestamp$
    if $B$ in $H$ then
        $RC \leftarrow RC + 1$
        $RAR(H(B).lt, CT - H(B).lt) = RC/CC$
        $H(B).lc \leftarrow CC$
        $H(B).lt \leftarrow CT$
    end
    else
        Initialize H($B$)
        $H(B).lc \leftarrow CC$
        $H(B).lt \leftarrow CT$
        Insert $H(B)$ into $H$
    end
    $update\_reuse\_distance(B)$
end
return $get\_miss\_ratio\_curve(RD)$
```

Algorithm 2: Subroutine *update_reuse_distance*

```
Input: currently accessed block $B$
if $B$ in $H$ then
    $time\_interval = CT - H(B).lt$
    $traffic = CC - H(B).lc$
    $rd(B) = (1 - RAR(H(B).lt, time\_interval)) * traffic$
    $RD(rd(B)) \leftarrow RD(rd(B)) + 1$
end
```

Algorithm 3: Subroutine *get_miss_ratio_curve*

```
Input: the reuse distance distribution $RD$
$total = sum(RD)$
$tmp = 0$
for $element$ in $RD$ do
    $tmp \leftarrow tmp + element$
    $MRC.append(1 - tmp/total)$
end
return $MRC$
```

Simulator Design. We have implemented a trace-driven simulator in C++ language for the rapid verification of the optimization strategy. The architecture of the simulator consists of an *I/O generator*, an *I/O router*, *cache instances* and *storage nodes*, etc. The *I/O generator* is for trace reading and transforming the trace records to the specific I/O structure of the simulator. The *I/O router* is responsible for request routing and forwarding, which is used to simulate the forwarding layer (shown in Fig. 1) to map each request to a specific storage node. The *storage nodes* simulate the nodes at the storage server layer (shown in Fig. 1). Each node is responsible for one magnetic storage drives and maintains the data mapping relationships inside that node. The *cache instances* is between the *I/O router* and the *storage nodes* and is part of the cache layer of the storage system. Each instance belongs to only one storage node and consists of the index map, metadata list, configuration structure, statistic housekeeping data structure, etc. The index map is implemented by using the *unordered_map* in C++ STL and the metadata list is organized according to the cache algorithm. Considering our cloud simulator is designed to be cloud storage system oriented, we choose only to use our own CBS trace in our evaluations. In our future work, we plan to evaluate our approach using other available traces, especially for comparing the efficacy of constructing MRCs.

4.2 Basic Comparisons

In this section we compare the cache model based on re-access ratio (hereafter called *RAR-CM*) with other three methods, including existing even-allocation method (*Original*), miniature simulation with the sampling idea from SHARDS [33] (*Mini-Simulation*), and an ideal case (*Ideal*) where exact miss ratio curves are used in placement of constructed cache models. We uses the jhash [35] function in implementing *Mini-Simulation* for the uniform randomized spatial sampling. This method leverages jhash to map each I/O record (using attributes like volume ID and data offset) to a location address. The accesses to the same physical block will be hashed to the same value. The I/O record will be selected only when $(V \ mod \ P) < T$, where P and T means the modulus and threshold, respectively. As in SHARDS, $SR = T/P$ represents the sampling ratio. In our experiments, we adopt a fixed sampling ratio of 0.01. We use the *RAR* curves in the prior 12 hours when calculating reuse distance. As illustrated in Fig. 9, the *RAR* curves exhibit good stability, i.e., they show minimum variations in the following days.

Table 2 shows the overall experimental results. In our configuration, we set the average cache size for each storage node as 200 GB (currently-practical configuration). All cache models perform comparably in terms of hit ratio. However, we have observed important back-end traffic savings despite of the seemingly negligible hit ratio improvements. *RAR-CM* compared to *Original* assignment policy with same amount of cache space reduces I/O traffic to back-end storage server by 13.2%. To achieve the same improvement, the *Original* method would require 50% additional cache space on each storage node (i.e., increase from 200 GB / Node to 300 GB / Node) based on the traces we collected from the production CBS system.

Table 2: The overall experimental results

	Hit Ratio	Back-end Traffic	Average Error	Extra Traffic
Original	94.45%	1	-	No
Mini-Simulation	94.85%	0.929	0.017	Yes
RAR-CM	95.14%	0.868	0.005	No
Ideal	95.49%	0.806	0	No

Note: The back-end traffic are normalized to that of *Original* method.

The hit ratio of *Mini-Simulation* is also quite high: 0.29% and 0.64% less than our cache model and the ideal model, respectively. This is consistent with the results in the earlier studies [33].

4.3 Miss Ratio Curves

We next take a closer look at the miss ratio curves of the three cache models. Fig. 10 shows the miss ratio curves of *RAR-CM* (the blue solid line with the cross), *Mini-Simulation* based on SHARDS (the green dotted line), and the exact simulation (the orange solid line). This figure shows the results of 20 randomly selected, but representative storage nodes. Other storage nodes have similar results. The cache space requirements vary among storage nodes and the curves of *RAR-CM* are closer to the curves of the exact simulation than that of *Mini-Simulation* in most cases. The advantage might be attributed to *RAR-CM* constructing the cache model based on the full set of trace and *Mini-Simulation* using spatial sampling causing some fidelity loss.

To evaluate the deviations of curves against the exact miss ratio curves, we report the metric of Mean Absolute Error (MAE) commonly used in evaluating cache models [33, 34]. In our experiments, we compute miss ratio curves at cache sizes 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 200, 300, 400 and 500 GB. Fig. 11 presents the MAE error distributions of *RAR-CM* and *Mini-Simulation* for the selected 20 storage nodes. The MAE averaged across all 20 storage nodes (labeled "Total") for *RAR-CM* is smaller than for Mini-Simulation: 0.005 vs 0.017, in addition to being smaller for each of the 17 out of the 20 nodes.

4.4 Overall Efficacy of OSCA

In this section, we compare the overall efficacy of *OSCA* in terms of hit ratio and backend traffic using the above mentioned three cache models, respectively. We present the results

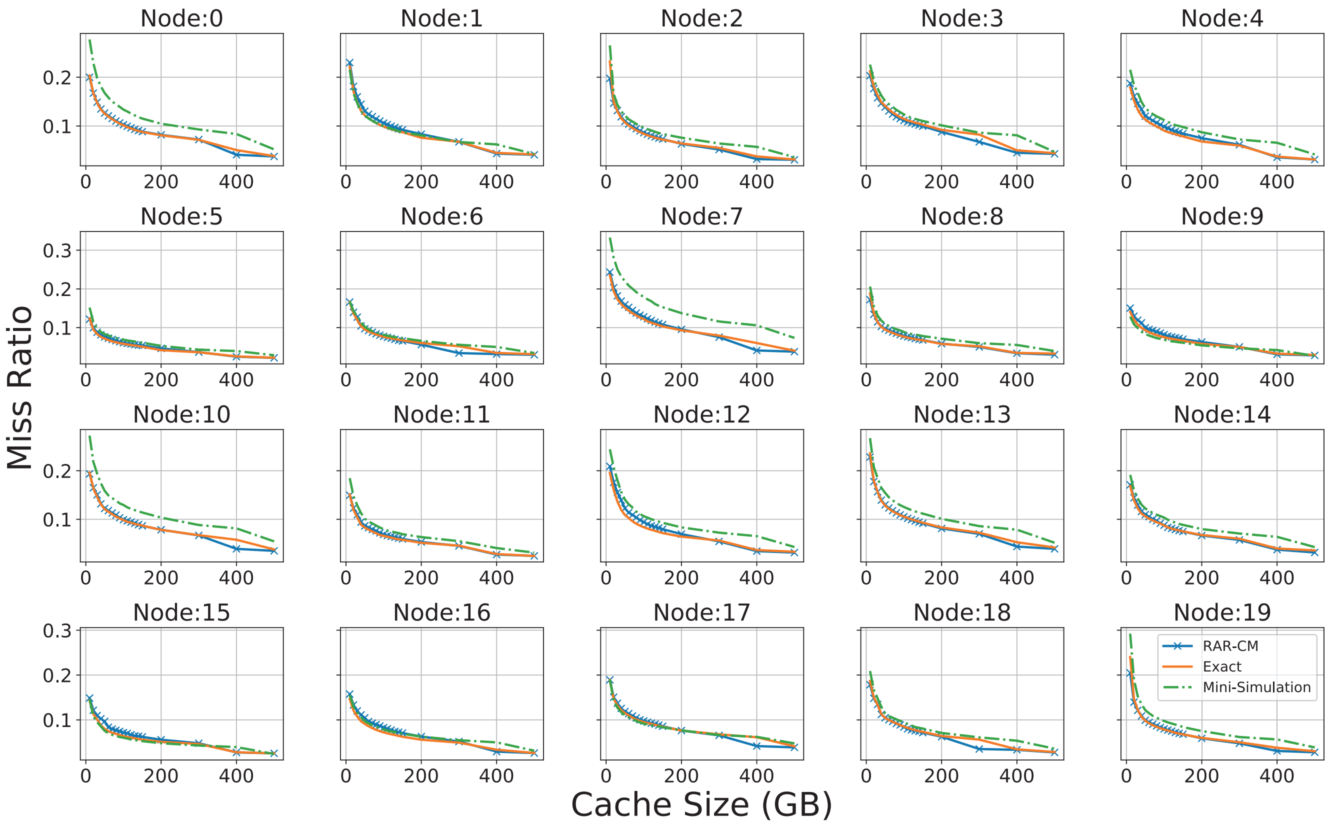


Figure 10: The miss ratio curve of 20 storage nodes. The cache space requirements vary among storage nodes and the curves of *RAR-CM* are closer to the curves of the exact simulation than that of *Mini-Simulation in most cases*.

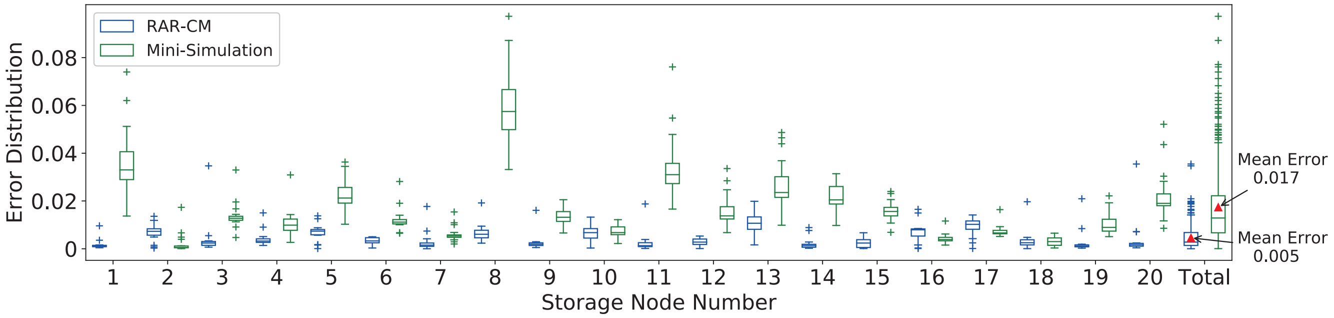
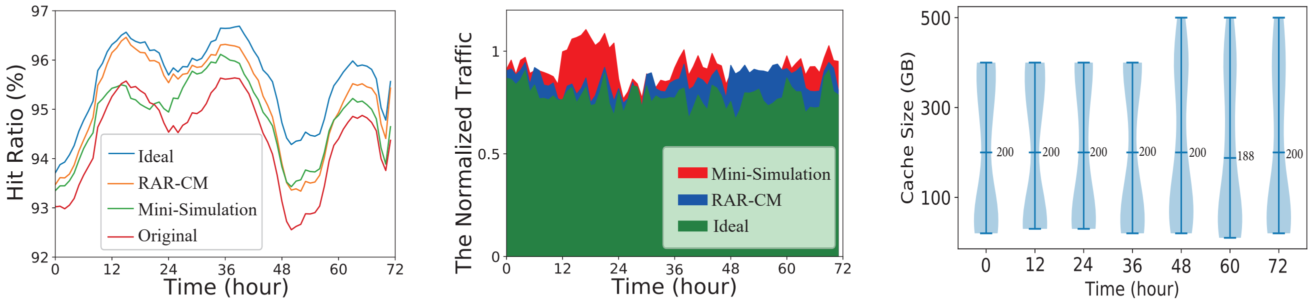


Figure 11: The MAE error distribution of our method *RAR-CM* and *Mini-Simulation* among storage nodes. The last two boxes are total MAE results. The middle lines in boxes indicate the middle values. The bottom and top side of the box represent the quartiles and the lines that extend out of the box (whiskers) represent data outside the upper and lower quartiles.

from the last three days of the trace, using the first 3 days as warm up periods. As shown in Fig. 12-a, *OSCA* based on *RAR-CM* can outperform the original assignment policy in the cache hit ratio without requiring additional cache space. Fig. 12-b shows the back-end traffic with different cache management policies. The back-end traffic is normalized to that of *Original* method. From the figure, we can know that on average, *OSCA* based on *RAR-CM* can reduce I/O traffic to back-end storage server by 13.2%. As shown in Fig. 12, *RAR-CM* results in slightly better hit ratios that *Mini-Simulation* except for hours $48-60$.

Fig. 12-c show the cache size configuration for each node at different times determined by our *OSCA* algorithm with *RAR-CM*. It can be seen that the demand for cache space varies considerably between nodes and our approach did respond correspondingly to meet the needs at different times.



(a) (b) (c)

Figure 12: Fig. (a) and Fig. (b) represents the hit ratio results for the last three days and the normalized back-end traffic using the three cache models, respectively. Fig.(c) shows OCSA adjusts the cache space for 20 storage nodes dynamically in response to their respective cache requirements decided by our cache modeling. The middle line in Fig. (c) represents the average cache size for each node. The results are obtained from traces mentioned in Section 4.1.

Based on the optimal cache size configuration scheme, *OSCA* periodically reassigns the corresponding cache size to each cache node every 12 hours.

4.5 Discussion

When trace collection and processing present a significant cost, *RAR-CM* offers an attractive alternative to other state-ot-the-art techniques. In this section, we make a comparison between *RAR-CM* and *Mini-Simulation* in terms of CPU, memory, network usage.

As mentioned in Section 3.2, upon each block request, *RAR-CM* first checks its history information in a hash map and calculates the block reuse distance. The history information of each referenced block contains two 64-bit numbers denoting the last access timestamp and the block sequence number of the last reference to each block, respectively. In our experiment, there are approximately 55.8 million unique blocks referenced each day in a storage node, occupying only 0.87 GB memory space via using *RAR-CM*. Besides the low memory resource usage, *RAR-CM* does not induce extra network traffic as all the computation is completed on the storage server nodes, enabling the miss ratio curves to be constructed and readily available in an online fashion. As for the CPU resource usage, as shown in Section 3.2, the reuse distance calculation of each block is very lightweight which only involves several simple operations and takes hundreds of nanoseconds.

Mini-Simulation needs to concurrently run multiple simulation instances to construct the cache miss ratio in different cache sizes. However, for very long traces, this method can consume a large number of computation resources (in our implementation, we start a thread in the main routine for each cache algorithm in a specific cache size). More importantly, I/O traces (there are about 4.46 billion I/O records per day in a typical CBS system) ought to be transmitted to and analyzed by a dedicated analysis system to avoid influencing service times. According to our experimental results, the transmission of the I/O records from these 20 nodes consumes approximately 72 GB of network bandwidth each day.

To quantify the runtime overhead, we have experimented with the *Mini-Simulation* algorithm. Specifically, we run 20 simulation routines (each routine starts 20 threads) simultaneously on a 12-core CPU (i.e., Intel Xeon CPU E5-2670 v3). The traces are stored in a storage server and each thread accesses the traces via the network file system. This method takes around 69 minutes to analyze a one-day-long I/O trace file and most of the time is consumed in trace reading (1.067 µs / record) and I/O mapping (2.406 µs / record). The I/O mapping determines which storage node should a record be assigned to based on the MD5 digest from the information of the record. We maintain the total time for the trace reading and I/O mapping and divide them by the total number of records processed to obtain the overhead per record.

5 Related Work

Our work is mostly related to the management of shared cache resource, which widely exists in various contexts, including multi-core processors, web applications, cloud computing and storage. A variety of methods have been proposed and they can be generally classified into heuristic methods, model-based quantitative methods.

Heuristic Methods: To achieve fairness in cache partitioning, the max-min fairness (MMF) and weighted max-min fairness methods are popularly used [12]. These two methods fairly satisfy the minimum requirements of each user and then evenly allocate unused resources to users having additional requirements. Different from MMF, Parihar et al. [25] propose the method of cache rationing, which ensures that the program cache space is not less than a set value and free cache space is allocated to a specific program. Kim, et al. [19] propose TCM which divides threads into delay-sensitive and bandwidth-sensitive groups and apply different cache policies to them. Similar to the TCM method, Zhuravlev et al. [46] proposed a scheduling algorithm called Distributed Intensity (DI), which adjusts the scheduling algorithm by analyzing the classification schemes of each thread through a novel methodology. Other methods, like [32], [12], and [42], have been proposed based on the game theory principles.

Model-based Quantitative Methods: Besides heuristic methods mentioned above, there have also been proposed many quantitative methods. These methods use locality metrics (e.g., Working Set Size, Average Footprint, Reuse Distance, and so on) to quantify the locality of the access patterns so as to predict the hit (or miss) ratio [7]. Reasonably, a shared-cache partition can be efficient using quantitative methods. **Working Set Size**. Inspired by the principle of locality, there are many studies [2, 9, 10] modeling the locality characteristics using working set size (WSS). For instance, based on the WSS theory, Arteaga et al. [2] propose an on-demand cloud cache management method. Specifically, they used Reused Working Set Size (RWSS) model, which only captures data with strong temporal locality, to denote the actual demand of each virtual machine (VM). Using the RWSS model, they can satisfy VM cache demand and slow down the wear-out of flash cache as well. **Footprint**. Footprint, which is defined as the number of unique data blocks referenced in a time interval, has been widely applied to cache resources allocation. Various methods have been proposed to estimate the footprint of workloads [6, 8, 28, 37] and they make trade-off between the complexity and accuracy of the measurement. Xiang et al. [38] propose the HOTL theory, which calculates the average footprint in a linear time complexity and apply the HOTL theory to transfer the average data footprint to reuse distance and predict the miss ratio in their following work [39]. By using this method, they can predict the interference of cache sharing without the need of parallel testing with multiple of cache sizes, and thus the miss ratio can be evaluated with low overhead. **Reuse Distance**. Reuse distance, defined as the unique accesses between two consecutive references to the same data, can be translated to hit ratio and a host of research efforts have been put to efficiently obtain reuse distance. Mattson et al. [22] give the definition of reuse distance and propose a specific method to measure reuse distance. Later researches use tree-based structure to optimize the computation complexity of reuse distance calculation [1, 5, 24, 43]. Waldspurger et al. [34] propose a spatially hashed approximate reuse distance sampling (SHARDS) algorithm to efficiently obtain reuse distance distribution and construct approximate miss rate curve. Hu et al. [14] propose the concept of average eviction time (AET) and relate the miss ratio at cache size c with AET using the formula mr(c) = P(AET(c)), which indicates that the miss ratio is the proportion of data whose reuse distance is greater than AET. In this study, AET is obtained through the Reuse Time Histogram (RTH) with a certain sampling method.

6 Conclusion

Cloud block storage (CBS) systems employ cache servers to improve the performance for cloud applications. Most existing cache management policies fall short of being applied to CBSs due to their high complexity and overhead, especially in the cloud context with large amount of I/O activity. In this paper, we propose a cache allocation scheme named *OSCA* based on a novel cache model leveraging re-access ratio. *OSCA* can search for a near optimal configuration scheme at a very low complexity. We have experimentally verify the efficacy of *OSCA* using trace-driven simulation with I/O traces collected from a production CBS system. Evaluation results show that *OSCA* offers lower MAE and computational and representational complexity compared with miniature simulation based on the main idea of SHARDS. The improvement in hit ratio leads to a reduction of I/O traffic to the back-end storage server by up to 13.2%. We are working on releasing our traces via the SNIA IOTTA repository [27] and integrating our proposed technique into the real CBS product system.

Acknowledgments

We would like to thank the anonymous reviewers for the valuable feedbacks and comments. We are especially grateful to our shepherds Jiri Shindler and Michael Mesnier for their tremendous help in improving the presentation and paper quality. We would also like to thank Tencent Technology (Shenzhen) Co., Ltd. for experimental environment, I/O trace support and releasing the trace to the community. This work is supported by the Innovation Group Project of the National Natural Science Foundation of China No.61821003.

References

[1] George Almási, Călin Caşcaval, and David A Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*, pages 37–43, 2002.

[2] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 355–369, 2016.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, pages 389–403, 2018.

[5] Bryan T Bennett and Vincent J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[6] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, pages 169–180, 2005.

[7] Daniel Byrne. A survey of miss-ratio curve construction techniques. *arXiv preprint arXiv:1804.01972*, 2018.

[8] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, pages 340–351. IEEE, 2005.

[9] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[10] Peter J Denning and Donald R Slutz. Generalized working sets for segment reference strings. *Communications of the ACM*, 21(9):750–759, 1978.

[11] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys '18)*, pages 1–13, 2018.

[12] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, volume 11, pages 24–24, 2011.

[13] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proceedings of the USENIX Annual Technical Conference (ATC '15)*, pages 57–69, 2015.

[14] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *Proceedings of the USENIX Annual Technical Conference (ATC '16)*, pages 351–364, 2016.

[15] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, 2013.

[16] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets '14)*, pages 1–7, 2014.

[17] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[18] Ke Zhou, Yu Zhang, et al. LEA: A lazy eviction algorithm for SSD cache in cloud block storage. In *Proceedings of the IEEE 36th International Conference on Computer Design (ICCD '18)*, pages 569–572, 2018.

[19] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 65–76, 2010.

[20] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 143–157, 2019.

[21] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[22] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[23] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, volume 3, pages 115–130, 2003.

[24] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. 1981.

[25] Raj Parihar, Jacob Brock, Chen Ding, and Michael C Huang. Protection and utilization in shared cache through rationing. In *Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT '14)*, pages 487–488, 2014.

[26] D Shasha and T Johnson. 2Q: A low overhead high performance buffer management replacement algoritm. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB '94)*, pages 439–450, 1994.

[27] SNIA. IOTTA. `http://iotta.snia.org/`.

[28] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 323–334, 2001.

[29] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. RapidMRC: approximating l2 miss rate curves on commodity systems for online optimizations. *ACM Sigplan Notices*, 44(3):121–132, 2009.

[30] Tencent. CBS. `https://intl.cloud.tencent.com/product/cbs`.

[31] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, pages 1–12. IEEE, 2016.

[32] Michail-Antisthenis I Tsompanas, Christoforos Kachris, and Georgios Ch Sirakoulis. Modeling cache memory utilization on multicore using common pool resource game on cellular automata. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(3):1–22, 2016.

[33] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of the USENIX Annual Technical Conference (ATC '17)*, pages 487–498, 2017.

[34] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 95–110, 2015.

[35] Wikipedia. Jhash. `https://en.wikipedia.org/wiki/Jenkins_hash_function`.

[36] Wolfram Mathworld. Weibull Distribution. `https://mathworld.wolfram.com/`.

[37] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. All-window profiling and composable models of cache sharing. *ACM SIGPLAN Notices*, 46(8):91–102, 2011.

[38] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*, pages 350–360. IEEE, 2011.

[39] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 343–356, 2013.

[40] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the Symposium on Cloud Computing (SOCC '17)*, pages 66–79, 2017.

[41] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *International Journal of Parallel Programming*, 45(1):30–44, 2017.

[42] Seyed Majid Zahedi and Benjamin C Lee. REF: Resource elasticity fairness with sharing incentives for multiprocessors. *ACM SIGPLAN Notices*, 49(4):145–160, 2014.

[43] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):1–39, 2009.

[44] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A Tencent case study. In *Proceedings of the International Conference on Supercomputing (ICS '18)*, pages 284–294, 2018.

[45] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

[46] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.

Lock-free Concurrent Level Hashing for Persistent Memory

Zhangyu Chen, Yu Hua, Bo Ding, Pengfei Zuo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract

With high memory density, non-volatility, and DRAM-scale latency, persistent memory (PM) is promising to improve the storage system performance. Hashing-based index structures have been widely used in storage systems to provide fast query services. Recent research proposes crash-consistent and write-efficient hashing indexes for PM. However, existing PM hashing schemes suffer from limited scalability due to expensive lock-based concurrency control, thus making multi-core parallel programing inefficient in PM. The coarse-grained locks used in hash table resizing and queries (i.e., search/insertion/update/deletion) exacerbate the contention. Moreover, the cache line flushes and memory fences for crash consistency in the critical path increase the latency. In order to address the lock contention for concurrent hashing indexes in PM, we propose *clevel* hashing, a lock-free concurrent level hashing, to deliver high performance with crash consistency. In the clevel hashing, we design a multi-level structure for concurrent resizing and queries. Resizing operations are performed by background threads without blocking concurrent queries. For concurrency control, atomic primitives are leveraged to enable lock-free search/insertion/update/deletion. We further propose context-aware schemes to guarantee the correctness of interleaved queries. Using real Intel Optane DC PMM, experimental results with real-world YCSB workloads show that clevel hashing obtains up to 4.2× speedup than the state-of-the-art PM hashing index.

1 Introduction

Non-volatile memory (NVM) deployed as persistent memory (PM) offers the salient features of large capacity, low latency, and real time crash recovery for storage systems [12, 30, 38]. Recently, Intel Optane DC persistent memory module (PMM) [2], the first commercial product of PM, is available on the market. Compared with DRAM, PM has 3× read latency and similar write latency [23,24,36]. In the meantime, the read and write bandwidths of PM achieve $1/3$ and $1/6$ of those of DRAM [23, 24, 27, 36]. PM delivers higher performance than SSD and the maximal 512 GB capacity for a single PM module is attractive for in-memory applications [23].

Building high-performance index structures for PM is important for large-scale storage systems to provide fast query services. Recent schemes propose some crash-consistent tree-based indexes, including NV-Tree [37], wB$^+$-Tree [14], FP-Tree [32], WORT [26], FAST&FAIR [22] and BzTree [9]. However, traversing through pointers in hierarchical trees hinders fast queries. Unlike tree-based schemes, hashing-based index structures leverage hash functions to locate data in flat space, thus enabling constant-scale point query performance. As a result, hash tables are widely used in many in-memory applications, e.g., redis [7] and memcached [4].

Existing hashing-based indexes for PM put many efforts in crash consistency and write optimizations but with little consideration for non-blocking resizing (also called rehashing) [27, 30, 40]. A hash function maps different keys into the same location, called hash collisions. In general, when the hash collisions can't be addressed or the *load factor* (the number of inserted items divided by the capacity) of a hash table approaches the predefined thresholds, the table needs to be expanded to increase the capacity. Traditional resizing operations acquire global locks and move all items from the old hash table to the new one. Level hashing [40] is a two-level write-optimized PM hashing index with cost-efficient resizing. The expansion of level hashing only rehashes items in the smaller level to a new level, which only migrates items in 1/3 buckets. However, the resizing operation in the level hashing is single-threaded and still requires a global lock to ensure correct concurrent executions. P-CLHT [27] is a crash consistent variant of Cache-Line Hash Table (CLHT) [17] converted by RECIPE [27]. The search operation in P-CLHT is lock-free, while the writes into stale buckets (all stored items have been rehashed) would be blocked until the full-table resizing completes. Hence, both schemes suffer from limited resizing performance, since the global lock for resizing blocks queries in other threads. Cacheline-Conscious Extendible Hashing (CCEH) [30], a persistent extendible hashing scheme, supports concurrent lock-based dynamic resizing, however coarse-grained locks for shared resources significantly increase the latency. Specifically, CCEH splits a segment, an array of 1024 slots by default, to increase the capacity, which requires the writer lock for the whole segment. Moreover, when the directory needs to be doubled, the global writer lock for directory is needed before doubling the directory. The Copy-on-Write (CoW) version of CCEH avoids the segment locks with the cost of extra writes due to the migration of inserted items. Hence, the insertion performance of CCEH with CoW is poorer than the default version with lazy deletion [30]. The concurrent_hash_map (cmap) in pmemkv [6] leverages lazy rehashing by amortizing data migration over future queries. However, the deferred rehashing may aggregate to a recursive execution in the critical path of queries, thus leading to non-deterministic query performance. Hence, current PM hashing indexes suffer from poor concurrency and scalability during resizing.

A scalable PM hashing index with concurrent queries is important to exploit the hardware resources and provide high throughput with low latency. Nowadays, a server node is able to provide tens of or even hundreds of threads, which enables the wide use of concurrent index structures. Existing hashing-based schemes for PM [27, 30, 40] leverage locks for inter-thread synchronization. However, coarse-grained exclusive locks in a critical path increase the query latency and decrease the concurrent throughput. Moreover, in terms of PM, the persist operations (e.g., logging, cache line flushes, and memory fences), when holding locks, further increase the waiting time of other threads. Fine-grained locks decrease the critical path but may generate frequent locking and unlocking for multiple shared resources. Moreover, the correctness guarantee is harder than coarse-grained locks.

In summary, in addition to crash consistency, we need to address the following challenges to build a high performance concurrent hashing index for PM.

1) Performance Degradation during Resizing. For concurrent hash tables, resizing operations need to be concurrently executed without blocking other threads. However, the resizing operation accesses and modifies the shared hash tables and metadata. Coarse-grained locks for global data ensure thread safety, but lead to high contention and significant performance degradation when the hash table starts resizing.

2) Poor Scalability for Lock-based Concurrency Control. Locking techniques have been widely used to control concurrent accesses to shared resources. The coarse-grained locks protect the hash table, but they also prevent concurrent accesses and limit the scalability. Moreover, the updates of shared data are often followed by flushing data into PM, which exacerbates lock contention. An efficient concurrent hashing scheme for PM needs to have low contention for high scalability while guarantee the concurrency correctness.

In order to address the above challenges, we propose *clevel hashing*, a crash-consistent and lock-free concurrent hash table for PM. Motivated by our level hashing [40], we further explore write-efficient open-addressing techniques to enable write-friendly and memory-efficient properties for PM in the context of concurrency. Different from the level hashing, our proposed clevel hashing aims to provide scalable performance and guarantee the correctness for concurrent executions. Unlike existing schemes [27, 30] that convert concurrent DRAM indexes to persistent ones, we propose a new and efficient way to enable persistent indexes to be concurrent with small overheads and high performance. Hence, the clevel hashing bridges the gap between scalability and PM efficiency.

To alleviate the performance degradation for resizing, we propose a dynamic multi-level index structure with asynchronous rehashing. Levels are dynamically added for resizing and removed when all stored items are migrated to a new level. The rehashing of items is offloaded into background threads, thus never blocking foreground queries. Background threads migrate the items from the last level to the first level via rehashing until there are two remaining levels. The two levels ensure a maximal load factor over 80% and the limited accesses to buckets for queries. Therefore, when rehashing is not running (the usual case for most workloads), the time complexity for search/insertion/update/deletion is constant-scale.

To provide high scalability with low latency, we design write-optimal insertion and lock-free concurrency control for search/insertion/update/deletion. The new items are inserted into empty slots without any data movements, hence ensuring write efficiency. For concurrent modifications to the hash table, clevel hashing exploits the atomicity of pointers and uses Compare-And-Swap (CAS) primitives for lock-free insertion, update, and deletion. Guaranteeing the correctness for lock-free queries with simultaneous resizing is challenging, since interleaved operations can be executed in any order and lead to failures and duplicate items. In the clevel hashing, we propose context-aware algorithms by detecting the metadata information changes to avoid inconsistencies for insertion, update, and deletion. The duplicate items are detected and properly fixed before modifying the hash table. In summary, we have made the following contributions in the clevel hashing.

- **Concurrent Resizing.** In order to address the bottleneck of resizing, we propose a dynamic multi-level structure and concurrent resizing without blocking other threads.
- **Lock-free Concurrency Control.** We design lock-free algorithms for all queries in the clevel hashing. The correctness for lock-free concurrency control is guaranteed with low overheads.
- **System Implementation.** We have implemented the clevel hashing using PMDK [5] and compared our proposed clevel hashing with state-of-the-art schemes on

real Intel Optane PM hardware. The evaluation results using YCSB workloads show the efficacy and efficiency of the clevel hashing. We have released the open-source code for public use.[1]

2 Background

2.1 Crash Consistency in Persistent Memory

Persistent memory (PM) provides the non-volatility for data stored in main memory, thus requiring crash consistency for data in PM. For store instructions, the typical maximal atomic CPU write size is 8 bytes. Therefore, when data size is larger than 8 bytes, system failures during sequential writes of data may lead to partial updates and inconsistency. In the meantime, the persist order of data in write-back caches is different from the issue order of store instructions, thus demanding memory barriers to enforce the consistency. To guarantee consistency, recent CPUs provide instructions for cache line flushes (e.g., *clflush*, *clflushopt*, and *clwb*) and memory barriers (e.g., *sfence*, *lfence*, and *mfence*) [1]. With these instructions, users can use logging or CoW to guarantee crash consistency for data larger than 8 bytes [27, 38]. However, logging and CoW generate extra writes causing the overheads for PM applications [30,40]. In our implementation, we use the interface provided by PMDK [5], which issues clwb and sfence instructions in our machine, to persist the data into PM.

2.2 Lock-free Concurrency Control

Compare-And-Swap (CAS) primitives and CoW have been widely used in existing lock-free algorithms for atomicity. The CAS primitive compares the stored contents with the expected contents. If the contents match, the stored contents are swapped with new values. Otherwise, the expected contents are updated with the stored contents (or just do nothing). The execution of CAS primitives is guaranteed to be atomic, thus avoiding the use of locks. CAS primitives are used in concurrent index structures to provide high scalability [21,34]. However, CAS primitives don't support data sizes larger than the CPU write unit size (e.g., 8 bytes). CoW is used to atomically update data larger than 8 bytes [30]. CoW first copies the data to be modified and performs in-place update in the copied data. Then the pointer is atomically updated with the pointer to new data using a CAS primitive. The drawback of CoW is the extra writes for the copy of unchanged contents. In PM, frequent use of CoW causes severe performance degradation [30, 40]. In our clevel hashing, we design the lock-free algorithms using CAS primitives for most writes and lightweight CoW for infrequent metadata updates, thus achieving high scalability with limited extra PM writes.

2.3 Basic Hash Tables

Unlike tree-based index structures, hashing-based indexes store the inserted items in flat structures, e.g., an array, thus obtaining $O(1)$ point query performance. Some hashing schemes, e.g., CLHT [17], store key-value items in the hash table, which mitigates the cache line accesses. However, such design doesn't support the storage of variable-length key-value items. Reserving large space in hash tables causes heavy space overheads, since most key-value pairs in real-world scenarios are smaller than a few hundreds of bytes [10, 18]. In order to efficiently support variable-length key-value items, many open-source key-value stores (e.g., redis [7], memcached [4], libcuckoo [28], and the cmap engine in pmemkv [6]) store pointers in hash tables and actual key-value items out of the table. In our clevel hashing, we store pointers in hash tables to support variable-length key-value items.

A typical hash table leverages hash functions to calculate the index of a key-value item. Different key-value items can be indexed to the same storage position, called *hash collisions*. Existing hashing schemes leverage some techniques to address hash collisions, e.g., linear probing [30], multi-slot buckets [18, 28, 30, 40], linked list [6, 16, 27, 29], and data relocation [18, 28, 34, 40]. If hash collisions cannot be addressed, the hash table needs to be resized to increase the capacity. Typical resizing operations consist of three steps: (1) Allocate a new hash table with 2× as many buckets as the old table. (2) Rehash items from the old table to the new table. (3) When all items in the old table have been rehashed, switch to the new table. The resizing in conventional hashing schemes involves intensive data movements [40] and blocks concurrent queries [30].

2.4 Hashing-based Index Structures for PM

Recently, researchers have proposed several hashing-based indexes for PM [6, 16, 30, 40]. Different from DRAM indexes, PM indexes need to remain consistent after system failures. However, the write bandwidth of PM is one sixth as much as DRAM [23, 24, 36], which indicates the significance of write efficiency for concurrent PM hashing indexes.

2.4.1 The Level Hashing Scheme

Our clevel hashing is based on the level hashing index structure [40]. Level hashing has three goals: low-overhead crash consistency, write efficiency, and resizing efficiency. Below, we briefly introduce the relevant components in level hashing.

Level hashing has two levels and the top level has twice the buckets of the bottom level. Each *level* is an array of 4-slot buckets. Besides the 4 slots, a bucket has 4 tokens and each token is one bit corresponding to one slot for crash

[1] https://github.com/chenzhangyu/Clevel-Hashing

Table 1: The Comparisons of Our Clevel Hashing with State-of-the-art Concurrent Resizable Hashing Indexes for PM. *(For abbreviation, "LEVEL" is the level hashing, "CCEH" is the default CCEH version using the MSB segment index and lazy deletion. "CMAP" is the concurrent_hash_map in pmemkv, and "CLEVEL" is our clevel hashing. For the memory efficiency and crash consistency, "✓" and "-" indicate good and moderate performance, respectively.)*

	Concurrency Control			Correctness Guarantee		Memory Efficiency	Crash Consistency
	Search	*Insertion/Update/Deletion*	*Resizing*	*Duplication*	*Missing*		
LEVEL	Slot lock	Slot lock	Global metadata lock	No	No	✓	✓
CCEH	Segment reader lock	Segment writer lock	Global directory lock	No	Yes	-	✓
CMAP	Bucket reader lock	Bucket writer lock	Bucket writer lock + lazy rehashing	Yes	Yes	✓	✓
P-CLHT	Lock-free	Bucket lock	Global metadata lock	Yes	Yes	✓	✓
CLEVEL	Lock-free	Lock-free	Asynchronous	Yes	Yes	✓	✓

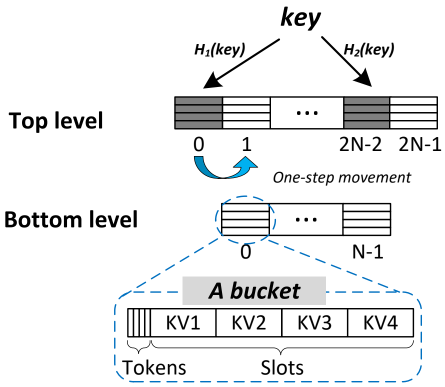


Figure 1: The level hashing index structure

consistency. The overview of level hashing index structure is shown in Figure 1.

By using two independent hash functions, each item has two candidate buckets in one level for storage (16 slots in total for two levels). When the two buckets are full, level hashing tries to perform one-step movement to obtain an empty slot for the item to be inserted. For example, the key in Figure 1 has two candidate buckets in the top level: the first bucket and the second to last bucket. If the two buckets are full and one stored item in the first bucket has empty slots in its alternative candidate bucket (e.g., the second bucket), the stored item is moved to the second bucket so that the key can be inserted to the first bucket. The one-step movement in level hashing improves the maximal load factor before resizing by 10% [40].

For resizing, level hashing creates a new level with 2× (e.g., 4N in Figure 1) as many buckets as the top level and migrates stored items in the bottom level to the new level. The items in the top level are reused without rehashing.

Level hashing uses slot-grained locks for concurrent queries. A fine-grained slot lock is acquired before accessing the corresponding slot and released after completing the access. For resizing, level hashing rehashes items using one thread and blocks concurrent queries of other threads. The concurrency control in level hashing has two correctness problems:

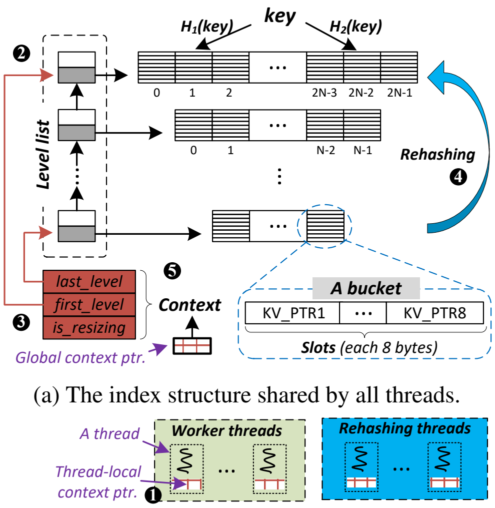
1) Duplicate items. An insertion thread with a single slot lock for an item cannot prevent other threads from inserting items with the same key into other candidate positions, since one item has 16 slots (2 candidate buckets for each level) for storage. Duplicate items in the hash table violate the correctness for updates and deletions: one thread updates or deletes one item while future queries may access the duplicate items that are unmodified.

2) Missing items. Items in level hashing are movable due to one-step movement and rehashing, while a slot lock cannot stop the movements. As a result, one query with a slot lock may miss inserted items due to concurrent moving of other threads.

2.4.2 Concurrent Hashing Indexes for PM

Recent schemes design some crash-consistent PM hashing indexes with lock-based concurrency control. CCEH [30] organizes 1024 slots as a segment for dynamic hashing. For concurrent execution, the segment splitting during insertion requires an exclusive writer lock for the segment. Moreover, when the number of segments reaches a predefined threshold, a global directory of segments needs to be doubled with a global directory lock. In pmemkv [6], there is a concurrent linked-list based hashing engine for PM, called cmap, which uses bucket-grained reader-writer locks for concurrency control. The cmap leverages lazy rehashing to amortize data migration in future queries. However, the aggregation of rehashing in the critical path of queries leads to uncertainty and increases the tail latency. P-CLHT [27] is a crash-consistent version of CLHT [17], a cache-efficient hash table with lock-free search. However, when P-CLHT starts resizing, concurrent insertions to the stale buckets (i.e., buckets whose items have been rehashed) have to wait until the resizing completes.

As shown in Table 1, we summarize state-of-the-art concurrent hashing-based index structures with resizing support for PM and compare our clevel hashing with them. All comparable schemes are the open-source versions with default parameter settings. For concurrent queries, CCEH uses coarse segment reader-writer locks, while level hashing and cmap adopt fine-grained locks. P-CLHT leverages bucket-grained



(a) The index structure shared by all threads.

(b) The local context for each thread.

Figure 2: The clevel hashing index overview.

locks for insertion/update/deletion and provides lock-free search. In terms of resizing, level hashing, CCEH, and P-CLHT suffer from the global locks. Though cmap avoids expensive global locks for resizing, the lazy rehashing is in the critical path of queries and affects the scalability. For concurrency correctness, as discussed in §2.4.1, level hashing suffers from duplicate items and missing inserted items. Since CCEH doesn't check if a key to be inserted is present in the hash table, CCEH also has the problem of duplicate items. For memory efficiency, CCEH sets a short linear probing distance (16 slots) by default to trade storage utilization for query performance. Unlike existing schemes, our clevel hashing achieves lock-free queries with asynchronous background resizing while guarantees the concurrency correctness and memory efficiency.

3 The Clevel Hashing Design

Our proposed clevel hashing leverages flexible data structures and lock-free concurrency control mechanism to mitigate the competition for the shared resources and improve the scalability. The design of clevel hashing aims to address the three problems in hashing index structures for PM: (1) ***How to support concurrent resizing operations without blocking the queries in other threads?*** (2) ***How to avoid lock contention in concurrent execution?*** (3) ***How to guarantee crash consistency with low overheads?*** In this Section, we first illustrate the dynamic multi-level structure (§3.1), which provides high memory efficiency and supports the low-cost resizing operation. We further present the lock-free concurrency control and correctness guarantee in the clevel hashing (§3.2), i.e., lock-free search/insertion/update/deletion. We finally discuss crash recovery (§3.3).

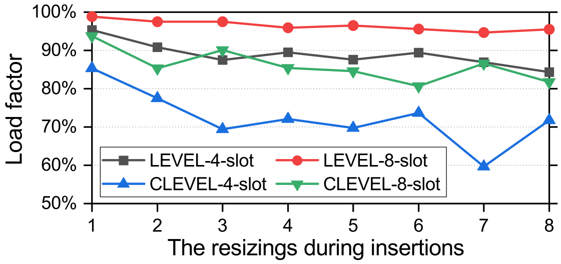


Figure 3: The load factors of level hashing and clevel hashing with different slots per bucket when the resizing occurs.

3.1 The Clevel Hashing Index Structure

3.1.1 Dynamic Multi-level Structure

The global index structure of clevel hashing shared by all threads is shown in Figure 2(a). The hash table in the clevel hashing consists of several levels and each level is an array of buckets. All these levels are organized by a linked list, called *level list*. For two adjacent levels, the upper level has 2× as many buckets as the lower one. The *first level* is interpreted as the level with the most buckets while the *last level* is interpreted as the level with the least buckets. Each key-value item is mapped to two candidate buckets in each level via two hash functions. To guarantee high storage utilization, clevel hashing maintains at least two levels [40]. Unlike the 4-slot bucket in the level hashing [40], each bucket in clevel hashing has 8 slots. Each slot consists of 8 bytes and stores a pointer to a key-value item. The actual key-value item is stored outside of the table via dynamic memory allocation. Hence, the 8-slot bucket is 64 bytes and fits the cache line size. By only storing the pointers to key-value items in slots, clevel hashing supports variable-length key-value items. Hence, the content in each slot can be modified using atomic primitives. The atomic visibility of pointers is one of the building blocks for lock-free concurrency control (§3.2) in our clevel hashing.

Different from the level hashing, our clevel hashing index structure is write-optimal for insertion while maintaining high storage utilization. The level hashing tries to perform one movement (one-step movement) for inserted items by copying them into their second candidate bucket in the same level, which causes one extra write for PM. For clevel hashing, the one-step movement is skipped, which decreases the storage utilization. Figure 3 shows the load factors of level hashing and clevel hashing with different slot numbers when successive resizings occur during insertion. Compared with level hashing having the same number of slots per bucket, the load factor of clevel hashing becomes lower due to the lack of one-step movement. However, 8-slot buckets in clevel hashing increase the number of candidate slots for a key in one level, thus achieving a comparable load factor (80%) than the level hashing with 4-slot buckets (default configuration). The number of slots per bucket also affects the throughput, which is discussed in §4.2.

3.1.2 The Support for Concurrent Resizing

Clevel hashing leverages the dynamic multi-level design and context to support concurrent resizing operations. The number of levels in clevel hashing is dynamic: levels are added for resizing and removed when rehashing completes. The *context* in clevel hashing is interpreted as an object containing two level list nodes and the `is_resizing` flag. The two nodes point to the first and last levels while the flag denotes if the table is being resized. There is a global pointer to the context (Figure 2(a)) and each thread maintains a thread-local copy of the context pointer (Figure 2(b)). Hence, the context can be atomically updated using CoW + CAS. Since the context size is 17 bytes (i.e., two pointers and one Boolean flag) and the context changes only when we add/remove a level, the CoW overheads of context are negligible for PM.

The resizing operation in clevel hashing updates the level list and the context. Specifically, when hash collisions can't be addressed, the resizing is performed in the following steps: (Step ❶) Make a local copy of the global pointer to context. (Step ❷) Dereference the local copy of context pointer, and append a new level with twice the buckets of the original first level to the level list using CAS. If the CAS fails, update the local copy of the context pointer and continue with the next step, since other threads have successfully added a new level. (Step ❸) Use CoW + CAS to update the global context by changing the first level to the new level (e.g., L_{new}) and setting `is_resizing` to true. When the CAS fails, check if the new first level's capacity is no smaller than L_{new} and the `is_resizing` is true: if so, update the local context pointer and continue; otherwise, retry the CoW + CAS with the `is_resizing` (true) and optional new level L_{new} (if the first level's capacity is smaller than L_{new}). (Step ❹) Rehash each item in the last level. The rehashing includes two steps: copy the item's pointer to a candidate bucket in the first level via CAS, and delete the pointer in the last level (without CAS, and the correctness for possible duplicate items is guaranteed in insertion §3.2.2 and update §3.2.3). If the CAS fails, find another empty slot. If no empty slot is found, go to step ❷ to expand the table. (Step ❺) When rehashing completes, update the last level and optional `is_resizing` (if only two levels remain after resizing) in the global context atomically using CoW + CAS. If the CAS fails, try again if the last level in current context is unmodified. The resizing workflow is shown in Figure 2. Note that the reasons for three possible CAS failures in resizing are different: the CAS failures in step ❷ come from the concurrent expansion of other threads, i.e., the step ❷ in other threads; the failures in steps ❸ and ❺ are due to the concurrent execution of these two steps (steps ❸ and ❺) in different threads. As a result, the strategies for corresponding CAS failures are different as presented above.

To mitigate the insertion performance degradation due to resizing, clevel hashing leverages background threads to enable asynchronous rehashing. Specifically, we divide the resizing into two stages, including *expansion* (steps ❶, ❷, and ❸) and *rehashing* (steps ❹ and ❺) *stages*. The time-consuming rehashing stage is offloaded into background threads, called *rehashing threads*. Rehashing threads continuously rehash items until there are two levels left. Therefore, when the table is not being resized, the queries in clevel hashing guarantee constant-scale time complexity. The threads serving query requests are called *worker threads*. When the hash collisions can not be addressed by worker threads, they perform the three steps in the expansion stage and then continue the queries. Since the main operations for table expansion are simple memory allocation and lightweight CoW for context (17 bytes), the expansion overheads are low. Moreover, there is no contention for locks during expansion. As a result, the resizing operation no longer blocks queries.

Rehashing performance can be improved by using multiple rehashing threads. To avoid contention for rehashing, a simple modular function is used to partition buckets into independent batches for rehashing threads. For example, if there are two rehashing threads, one thread rehashes odd-number buckets while the other rehashes even-number buckets. After both rehashing threads finish, they update the global context following step ❺.

3.2 Lock-free Concurrency Control

In order to mitigate the contention for shared resources, we propose lock-free algorithms for all queries, i.e., search, insertion, update, and deletion.

3.2.1 Search

The search operation needs to iteratively check possible buckets to find the first key-value item that matches the key. There are two main problems for lock-free search in the clevel hashing: (1) *High read latency for the pointer dereference costs.* Since clevel hashing only stores the pointers in hash tables to support variable-length items, dereferencing is needed to fetch the corresponding key, which results in high cache miss ratios and extra PM reads. (2) *Missing inserted items due to the data movement.* The concurrent resizing moves the items in the last level, and therefore, searching without any locks may miss inserted items.

For the pointer dereference overheads, our proposed clevel hashing leverages a summary tag to avoid the unnecessary reads for full keys. A tag is the summary for a full key, e.g., the leading two bytes of the key's hash value. The hash value is obtained when calculating the candidate buckets via hash functions (e.g., the `std::hash` from C++ [8]). The tag technique is inspired from MemC3 [18] and we add atomicity for the pair of tag and pointer. For each inserted item, its tag is stored in the table. For a search request, only when the tag of a request matches the stored tag of an item, we fetch the stored full key by pointer dereferencing and compare the two keys. A *false positive* case for tags appears when different keys have the same tag. For 16-bit tags, the false positive rate is $1/2^{16}$. Since we check full-size keys when two tags match, the false positives can be identified and will not cause any problem of correctness. Instead of allocating additional space for tags in MemC3, clevel hashing leverages the unused 16 highest bits in pointers to store the tags. Current pointers only consume 48 bits on x86_64, thus leaving 16 bits unused in 64-bit pointers [31,35]. The reuse of reserved bits enables the atomic updates of pointers and tags.

To address the problem of missing inserted items, we propose to search from the last level to the first level, called *bottom-to-top* (*b2t*) search strategy. The intuition behind b2t searching is to follow the direction of bottom-to-top data movement in hash table expansion, which moves items from the last level to the first level. However, a rare case for missing is: after a search operation starts, other threads add a new level through expansion and rehashing threads move the item that matches the key of the search to the new level. To fix this missing, clevel hashing leverages the atomicity of context. Specifically, when no matched item is found after b2t search, clevel hashing checks the global context pointer with the previous local copy. If the two pointers are different, redo the search. The overheads for the re-execution of search are low, since changes of the context pointer are rare, occurring only when a level is added to or removed from the level list. Therefore, the correctness for the lock-free search is guaranteed with low costs.

3.2.2 Insertion

For insertion, a key-value item is inserted if the key does not exist in the table. The insertion first executes lock-free b2t search (§3.2.1) to determine if an item with the same key exists. If none exists, the pointer (with its summary tag) to the key-value item is atomically inserted into a less-loaded candidate bucket. When there is no empty slot, we resize the table by adding a new level with background rehashing (§3.1.2) and redo the insertion. However, lock-free insertion leads to two correctness problems: (1) *Duplicate items from concurrent insertions*. Without locks, concurrent threads may insert items with the same key into different slots, which results in failures for update and deletion. (2) *Loss of new items inserted to the last level*. When new items are inserted into the buckets in the last level that have been processed by rehashing threads, these inserted items are lost after we reclaim the last level.

For concurrent insertions to different slots, it is challenging to avoid the duplication in a lock-free manner, since atomic primitives only guarantee the atomicity of 8 bytes. However, each one of the duplicate items is correct. Hence, we fix the duplication in future updates(§3.2.3) and deletions(§3.2.4).

In order to fix the loss of new items, we design a context-aware insertion scheme to guarantee the correctness of insertion. The context-aware scheme includes two strategies: (1) Before the insertion, we check the global context and do not insert items into the last level when the hash table is resizing, i.e., when the `is_resizing` is true. (2) After the insertion, if the table starts resizing and the item has been already inserted into the last level, we redo the insertion using the same pointer without checking duplicate items. The re-execution of insertion leads to possible duplicate pointers in the hash table. However, duplicate pointers don't affect the correctness of search, because they refer to the same key-value items. Future updates and deletions are able to detect and address the duplication.

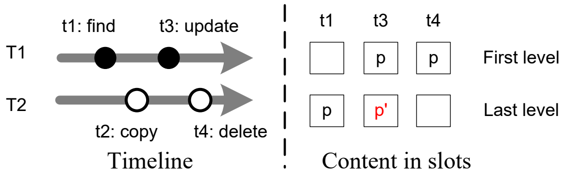


Figure 4: The update failure. (*"T1": an update thread, "T2": a rehashing thread, "t1-t4": timestamps, "p": pointer to the old item, "p′": pointer to the updated item.*)

3.2.3 Update

The update operation in the clevel hashing atomically updates the pointers to the matched key-value items. Different from the insertion, the update needs to fix duplicate items. Otherwise, duplicate items may lead to inconsistency after being updated. Moreover, the concurrent executions of resizing and update may cause update failures due to the data movement for rehashing. This section focuses on our solutions for the two correctness problems.

1) Content-conscious Find to Fix Duplicate Items. There are three cases for duplicate items in our clevel hashing: concurrent insertion with the same key, the retry of the context-aware insertion, and data movement for rehashing. Note that re-insertion after system crash would not generate duplication due to checking of the key before insertion. We observe that duplication from two concurrent insertions leads to two pointers to different items and keeping any one of the two is acceptable. Re-insertion or rehashing generates two pointers to the same item. In this case, we keep the pointer which is closer to the first level, since rehashing threads may delete the pointer in the last level. If two pointers are in the same level, keeping either pointer is identical. With this knowledge, we design a content-conscious Find process to handle duplication in two steps. First, we apply b2t search to find two slots storing the pointer to the matched key. Second, if two pointers refer to different locations, we delete the item and the pointer that first occurs in the b2t search. If two pointers point to the same location, we simply delete the pointer that first occurs. By removing duplicate items, the Find process returns at most one item for the atomic update.

2) Rehashing-aware Scheme to Avoid Update Failures. As the example shown in Figure 4, even with the Find process, the interleaved execution of update and rehashing is possible to lose the updated values. The updated item referred by p′ is deleted by the rehashing thread. A straightforward solution is to issue another Find process after the update. However, frequent two-round Find increases the update latency. To decrease the frequency, we design a rehashing-aware scheme by checking the rehashing progress. Specifically, before the first Find process, we record the bucket index (e.g., RB_{idx}) being rehashed by one rehashing thread. After the atomic update, we read the rehashing progress again (e.g., RB'_{idx}). If the global context doesn't change, an additional Find is triggered only when meeting the following constraints: (1) the table is during resizing; (2) the updated bucket is in the last level; (3) the updated bucket index B_{idx} satisfies $RB_{idx} \leq B_{idx} \leq RB'_{idx}$ for all rehashing threads. Since it's a rare case that three constraints are simultaneously satisfied, our rehashing-aware update scheme guarantees the correctness with low overheads.

3.2.4 Deletion

For deletion, clevel hashing atomically deletes the pointers and matched items. Like the update, the deletion also needs to remove duplicate items. Compared with the update, we optimize the scheme to handle duplication in deletion. Briefly speaking, clevel hashing deletes all the matched items through the b2t search. The lock-free deletion also has failures like the lock-free update. Hence, we use the rehashing-aware scheme presented in lock-free update(§3.2.3) to guarantee the correctness.

3.3 Recovery

The fast recovery requires the guarantee for crash consistency, which is nontrivial for PM. Recent studies [16, 35] show that a crash-consistent lock-free PM index needs to persist after stores and not modify PM until all dependent contents are persisted. The crash consistency guarantee in clevel hashing follows this methodology. Specifically, clevel hashing adds cache line flushes and memory fences after each store and persists dependent metadata, e.g., the global context pointer, after the load in insertion/update/deletion. The persist overheads for crash consistency can be further optimized by persisting in batches [16].

For the crash consistency in rehashing, clevel hashing records the index of bucket (e.g., RB_{idx}) in PM after successfully rehashing the items in the bucket. To recover from failures, rehashing threads read the context and bucket index and continue the rehashing with the next bucket (e.g., $RB_{idx}+n$, n is the number of rehashing threads). A crash during the data movement of rehashing may lead to duplicate items, which are fixed in future update (§3.2.3) and deletion (§3.2.4).

To avoid permanent memory leakage [16], we leverage existing PM atomic allocators from PMDK [5] and design lock-free persistent buffers for secure and efficient memory management. The PM atomic allocators atomically allocate and reclaim memory to avoid expensive transactional memory management [5]. A persistent buffer is a global array of persistent pointers (used by the PM allocators for atomic and durable memory management) attached to the root object (an anchor) of the persistent memory pool. The array size is equal to the thread number. Each thread uses the persistent pointer corresponding to its thread ID. Hence, there is no contention for the persistent buffers. When recovering from failures, we scan the persistent buffers and release the unused memory.

4 Performance Evaluation

4.1 Experimental Setup

Our experiments run on a server equipped with six Intel Optane DC PMM (1.5 TB in total), 128 GB DRAM, and 24.75 MB L3 cache. The Optane DC PMMs are configured in the *App Direct* mode and mounted with ext4-DAX file system. There are 2 CPU sockets (i.e., NUMA nodes) in the server and each socket has 36 threads. For a processor in one NUMA node, the latency of accessing local memory (attached to the NUMA node) is lower than non-local memory [13, 25, 33]. Conducting experiments across multiple NUMA nodes introduces the disturbance of non-uniform memory latencies. To avoid the impact of NUMA architectures, we perform all the experiments on one CPU socket by pinning threads to an NUMA node for all schemes, like RECIPE [27]. Existing NUMA optimizations, e.g., Node-Replication [13] to maintain per-node replicas of hash tables and synchronize these replicas through a shared log, are possible to improve the scalability with more NUMA nodes.

In our evaluation, we compare the following concurrent hashing-based index structures for PM:

- **LEVEL**: This is the original concurrent level hashing [40] with consistency support. The level hashing uses slot-grained reader-writer lock for queries and a global resizing lock for resizing.

- **CCEH**: CCEH [30] organizes an array of slots as a segment (e.g., 1024 slots) and uses a directory as an address table. Linear probing (e.g., 16 slots) is used to improve the load factor. CCEH supports dynamic resizing through segment splitting and possible directory doubling. We adopt the default lazy deletion version since it has higher insertion throughput than the CoW version. CCEH uses reader-writer locks for segments and the directory.

- **CMAP**: The concurrent_hash_map storage engine in pmemkv [6] is a linked list based concurrent hashing scheme for PM. It uses reader-writer locks for concurrent accesses to buckets and supports lazy rehashing (rehash the buckets in a linked list when accessing).

Table 2: Workloads from YCSB for macro-benchmarks.

Workload	Read ratio (%)	Write ratio (%)
Load A	0	100
A	50	50
B	95	5
C	100	0

- **P-CLHT**: P-CLHT [27] is a linked list based cache-efficient hash table. Each bucket has 3 slots. P-CLHT supports lock-free search while the bucket-grained lock is needed for insertion and deletion. The resizing in P-CLHT requires a global lock. When one thread starts rehashing, another helper thread (one helper at most) is allowed to perform concurrent rehashing, which is called helping resizing. The helping resizing mechanism is enabled by default.

- **CLEVEL**: This is our proposed scheme, clevel hashing, which provides asynchronous resizing and lock-free concurrency control for all queries with high memory efficiency.

Since open-source cmap is implemented using PMDK with C++ bindings [5], we implement our clevel hashing with PMDK (version 1.6) and port level hashing, CCEH, and P-CLHT to the same platform for fair comparisons. Like cmap and clevel hashing, we optimize level hashing, CCEH, and P-CLHT to support variable-length items by storing pointers in the hash table. For level hashing and CCEH, we use the same type of reader/writer locks from cmap to avoid the disturbance of lock implementations. During the porting, in addition to the reported bugs of the inconsistencies in directory metadata [27], we observe a concurrent bug for the directory in original CCEH: a thread performing search can access a directory deleted by other threads that are doubling the directory. As a result, failures may occur in search when accessing the reclaimed directory via pointer dereferencing. To ensure the correctness and avoid such failures in experiments, we add the missing reader lock for the directory. The hash functions for all schemes are the same: the `std::hash` from the C++ Standard Template Library (STL) [8]. In addition to conventional locks, we also evaluate the performance of level hashing, CCEH, and cmap with the spinlocks from Intel TBB library [3]. For abbreviation, *LEVEL-TBB*, *CCEH-TBB*, and *CMAP-TBB* are TBB-enabled.

We use YCSB [15] to generate micro-benchmarks in zipfian distribution with default 0.99 skewness [40] to evaluate the throughput of different slot numbers and latencies of different queries. The results using uniformly distributed workloads are similar due to the randomness of hash functions [30]. Different queries are executed in the micro-benchmarks: insertion (unique keys), positive search (queried keys exist), negative search (queried keys not exist), update, and deletion. The items to be updated or deleted are present in the table. To evaluate the concurrent throughput, we leverage the real-world workloads from YCSB as macro-benchmarks, following RECIPE [27]. The workload patterns are described in Table 2. We initialize all indexes with similar capacity (64 thousand for micro-benchmarks and 256 thousand for macro-benchmarks) and use 15-byte keys and 15-byte values for all experiments. The experiment with YCSB workloads consists of two phases: load and run phases. In the load phase, indexes are populated with 16 million and 64 million items for micro- and macro-benchmarks, respectively. In the run phase, there are 16 million queries for micro-benchmarks and 64 million for macro-benchmarks. For concurrent execution, each scheme has the same number of threads in total. During our evaluation of clevel hashing, we observe that one rehashing thread for 35 insertion threads can guarantee the number of levels is under 4. Hence, we set one thread as the rehashing thread by default. The reported latency and throughput are the average values of 5 runs.

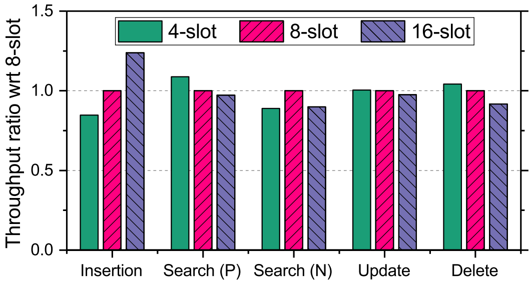


Figure 5: The normalized concurrent throughput of clevel hashing with different slots per bucket. *("Search (P)" is positive search and "Search (N)" is negative search.)*

4.2 Different Slot Numbers and Load Factor

In clevel hashing, the slots per bucket affects not only memory efficiency but also concurrent performance. We run the micro-benchmarks with different slot numbers in clevel hashing and measure the concurrent throughput with 36 threads. The throughput is normalized to that of an 8-slot bucket, as shown in Figure 5. With the increase of slots, the insertion throughput increases. The reason is that more slots per bucket indicate more candidate positions for a key so that it's easier to find an empty slot without resizing. Decreasing the slots per bucket reduces the number of slots to be checked and cache line accesses (a 16-slot bucket requires two cache lines), thus improving the search, update, and deletion throughputs. According to the results shown in Figure 5, 8-slot bucket is a trade-off between 4-slot and 16-slot buckets. Therefore, we set the slot number to 8.

In order to evaluate the memory efficiency of different schemes, we use an insert-only workload to record the load factor (the number of inserted items divided by the number of slots in the table) after every 10K insertions. Since the slot in cmap is allocated on demand for insertions, the load factor is always 100%. The load factors of the other schemes

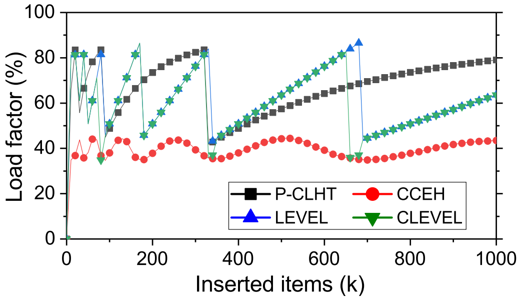


Figure 6: The load factor per 10K insertions. *(The even symbols are skipped for clearness.)*

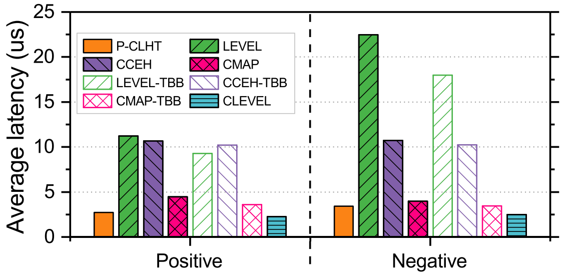


Figure 7: The average latency for concurrent search.

are shown in Figure 6. The maximal load factor of CCEH is no more than 45%, because CCEH probes only 16 slots to address the hash collisions. Though CCEH is able to increase the linear probing distance for higher memory efficiency, long probing distance leads to more memory accesses and pointer dereferencing, thus decreasing the throughputs for all queries. P-CLHT resizes when the number of inserted items approaches the initial capacity of current hash table. By using the three-slot bucket with linked list, the load factor of P-CLHT is up to 84%. Compared with level hashing, clevel hashing doesn't move items in the same level. However, clevel hashing increases the number of slots per bucket to 8. As a result, the maximal load factor of clevel hashing is comparable with original level hashing, i.e., 86%.

4.3 Micro-benchmarks

We use the micro-benchmarks to evaluate the average query latencies in different PM hashing indexes. The latency of a query is interpreted as the time for executing the query, not including the time waiting for execution. All experiments run with 36 threads. Note that the latencies of micro-benchmarks for search, update, and deletion demonstrate the performance of corresponding queries without the impact on resizing, since there is no insertion in these workloads. For the insert-only workload, the expansion of hash table occurs in the run phase.

For the concurrent search, we measure the average latencies when all keys exist (positive search) or not (negative search) in the table. As shown in Figure 7, the level hashing suffers from frequent locking and unlocking of candidate slots, especially

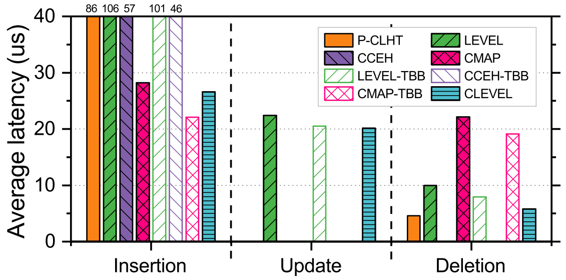


Figure 8: The average latencies for concurrent insertion, update, and deletion.

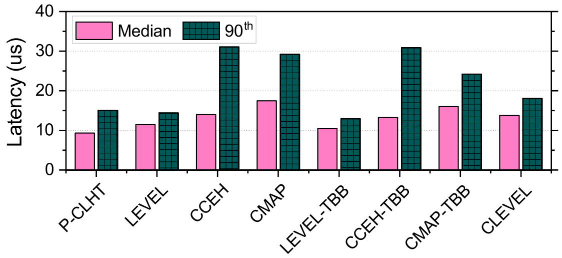


Figure 9: The median and 90^{th} percentile latencies for concurrent insertion.

for the negative search, i.e., 16 slot locks in total for 4 candidate 4-slot buckets. The coarse-grained segment lock (1024 slots) in CCEH leads to high search latency. The search in cmap only requires one bucket lock and ensures low latency. Due to the lock-free search, P-CLHT achieves lower latency than lock-based indexes. For clevel hashing, there are only two levels when the table is not resizing, thus ensuring the number of candidate buckets to be checked is at most 4. Moreover, the lock-free search avoids the contention for buckets. Tags filter unnecessary retrievals for keys. As a result, clevel hashing achieves 1.2×−5.0× speedup for positive search latency and 1.4×−9.0× speedup for negative search latency, compared with other PM hashing indexes.

The average latencies for insertion/update/deletion are shown in Figure 8. Some bars are missing because the corresponding schemes haven't implemented update (i.e., P-CLHT, cmap, and CCEH) or deletion (i.e., CCEH) in their open-source code.

Insertion: During insertion, all schemes have to expand to accommodate 16 million items. The resizing may block several requests (the number depends on the resizing times and thread numbers) and significantly increase their execution time, thus increasing the average latencies. P-CLHT, level hashing, and CCEH suffer from the global lock for resizing. By amortizing the rehashing over future queries, cmap achieves low average latency for insertions. The average latency of clevel hashing is slightly higher than the cmap with TBB because the expansion needs to durably allocate a large new level via the persistent allocator from PMDK, which is achieved by expensive undo logging [38].

Figure 9 shows the median and 90^{th} percentile insertion

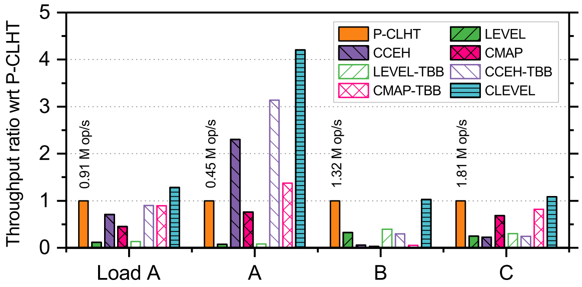


Figure 10: The concurrent throughput of YCSB normalized to P-CLHT.

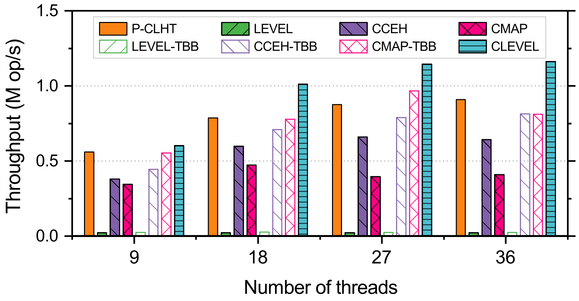


Figure 11: The insertion scalability.

latencies. Unlike the average latency, median and 90^{th} percentile latencies demonstrate the insertion performance without the impact of resizing. The reason is that the ratio of insertions which encounter resizing during their executions is less than 10%. In the meantime, the queuing time for execution is not included in the latency. CCEH suffers from high percentile latencies due to the coarse-grained segment lock. Though cmap leverages fine-grained bucket locks, the amortized rehashing in queries increases the insertion time. Due to the context-aware insertion for correctness guarantee, the median and 90^{th} percentile latencies of clevel hashing are slightly higher than level hashing and P-CLHT but lower than CCEH and cmap.

Update: Compared with original level hashing, clevel hashing obtains slightly lower update latency. The reason is that the benefits of lock-free update compensate for the overheads of correctness guarantee in the clevel hashing, e.g., additional Find operation for duplicate items and checking for update failures.

Deletion: The deletion latency in cmap is higher than other schemes due to rehashing the bucket if necessary before accessing. Level hashing has higher deletion latency than P-CLHT and clevel hashing due to the frequent locking and unlocking when accessing candidate slots. To fix duplication during deletion, clevel hashing checks all candidate slots, thus resulting in a slightly higher latency than P-CLHT.

4.4 Macro-benchmarks

Figure 10 shows the concurrent throughput normalized to P-CLHT of different PM hashing schemes with real-world workloads from YCSB. We run the experiment with 36 threads for all schemes. Since workload Load A is used to populate indexes with 64 million items in the load phase, all indexes resize multiple times (e.g., more than 10 times in clevel hashing) from small sizes. Resizing also occurs in the workload A.

The locks used for concurrency control hinder the index performance. Specifically, the global resizing lock in the level hashing blocks all queries until the single-threaded rehashing completes, which leads to low throughput in workload Load A and A. The global directory lock in CCEH is only used for directory doubling. Therefore, the insertion throughput is much higher than level hashing. Due to the helping mechanism in P-CLHT, there are two threads concurrently rehashing items, which mitigates the overheads of the global resizing lock in the load phase of YCSB (Load A). However, when the table size increases, the two threads are not enough to rehash all items in a short time, which accounts for the low throughput for P-CLHT in workload A. Due to the multiple resizing, the aggregated rehashing hinders the throughput of cmap. Unlike these lock-based indexes, the lock-free concurrency control in clevel hashing avoids the lock contention during insertions. Hence, clevel hashing obtains 1.4× speedup than cmap for insertion throughput. In summary, our clevel hashing achieves up to 4.2× speedup than the state-of-the-art PM hashing index (i.e., P-CLHT).

To evaluate the scalability of clevel hashing, we measure the insertion throughput with different number of threads using the Load A. As shown in Figure 11, with the increase of threads, the throughput of clevel hashing increases and is consistently higher than other schemes. This trend in search throughput is similar.

4.5 Discussion

The reduction of hash table size. The current design of clevel hashing doesn't support the reduction of the hash table size. When most stored items in the hash table are deleted, the table may reduce the table size to improve space utilization. Clevel hashing needs to be adapted to support the reduction. Specifically, to reduce the table size in clevel hashing, we need to create a new level with half of the buckets in the last level and rehash the items from the first level to the last level. When all the items of the first level are rehashed, the first level is reclaimed. The migration of items for the reduction generates data movement from the top level to the bottom one (i.e., top-to-bottom movement), which is opposite to expansion (i.e., bottom-to-top movement). Therefore, to support concurrent reduction, we carry out top-down search strategy instead of down-top search (§3.2.1) to avoid missing inserted items. Note that all threads need to leverage the same search strategy: either top-down searching for reduction or down-top searching for expansion. The clevel hashing with non-blocking concurrent reduction is our future work.

The isolation level. For the isolation in transactions, clevel hashing has dirty reads, since there is no lock to isolate data. Hence, the isolation level is *read uncommitted* [11]. To support higher isolation levels in a transaction, additional locks or version control schemes are required [11, 22].

Space overhead. The metadata overhead in clevel hashing mainly comes from the persistent buffers (§3.3), which are the arrays of persistent pointers to the allocated memory, for efficient management without contention. Persistent buffers have separate entries for each thread. Therefore, the metadata overhead is proportional to the number of threads. Clevel hashing achieves a maximal load factor over 80% before resizing. During resizing, hashing collisions that cannot be addressed increase the number of levels to more than 3. Due to the randomness of hash functions, the possibility of continuous hash collisions for one position is very low. Moreover, rehashing threads migrate items in the last level until only two levels remain. Hence, the number of total levels is usually small, which enables high storage utilization.

5 Related Work

5.1 Hashing-based Index Structures for PM

In order to optimize the hashing performance on PM, recent work have designed some hashing-based index structures for PM. Path hashing [39] and level hashing [40] leverage sharing-based index structures and write-efficient open-addressing techniques for high memory efficiency with limited extra writes. Level hashing introduces a cost-efficient resizing scheme by only rehashing the items in the old bottom level to the new top level, which only account for 1/3 of total inserted items. LF-HT [16] uses lock-free linked list for each bucket to address hash collisions. However, these three schemes all suffer from poor resizing performance, since the resizing operations require exclusive global lock for metadata. CCEH [30] is based on extendible hashing, which dynamically expands the hash table by segment splitting and optional directory doubling. Although the resizing in CCEH is concurrent with other queries, the use of coarse-grained locks for segments or even directory during resizing causes performance degradation. The cmap storage engine in pmemkv [6] supports concurrent lazy rehashing. The rehashing of items in a bucket is trigger when accessing the bucket. As a result, cmap distributes the rehashing of items to future search/insertion/update/deletion. However, cmap is possible to encounter recursive rehashing due to the lazy rehashing. The rehashing in the critical path of queries decreases the throughputs, especially for search operations. Unlike existing schemes, clevel hashing has dynamical multi-level structure for concurrent asynchronous resizing and designs lock-free algorithms to improve the scalability with low latency.

5.2 Lock-free Concurrent Hashing Indexes

Lock-free algorithms mitigate the lock contention for shared resources and are hard to design because of the challenging concurrency control. The lock-free linked list proposed by Harris [21] is widely used in lock-free concurrent hashing indexes [16, 29]. This class of schemes add a lock-free linked list to each bucket. Though the lock-free linked list enables lock-free insertion in these hash tables, it causes high search overheads due to the sequential iteration over linked lists. The lock-free cuckoo hashing [31] uses marking techniques with helping mechanism to support lock-free cuckoo displacements. However, the recursive data movements bring lots of extra PM writes [40]. Recent work [20] uses PSim [19] to build a wait-free resizable hash table. The wait-free technique relies on copying the shared object and helping mechanism, which still leads to extra writes on PM and introduces overheads due to helping. Moreover, the extendible hashing structures are memory inefficient as shown in our evaluation. Different from existing lock-free hashing schemes built on DRAM, clevel hashing designs PM friendly and memory efficient multi-level structures with simple but effective context-aware mechanism to guarantee correctness and crash consistency.

6 Conclusion

Persistent memory offers opportunities to improve the performance of storage systems, but suffers from the lack of efficient and concurrent index structures. Existing PM-friendly hashing indexes only focus on the consistency and write reduction, which overlook the concurrency and resizing of hash tables. In this paper, we propose clevel hashing, a lock-free concurrent hashing scheme for PM. Clevel hashing leverages the dynamic memory-efficient multi-level design and asynchronous resizing to address the blocking issue due to resizing. The lock-free concurrency control avoids the lock contention for all queries while guarantees the correctness. Our results using Intel Optane DC PMM demonstrate that clevel hashing achieves higher concurrent throughput with lower latency than state-of-the-art hashing indexes for PM.

Acknowledgments

This work was supported in part by National Key Research and Development Program of China under Grant 2016YFB1000202, National Natural Science Foundation of China (NSFC) under Grant No. 61772212 and No. 61821003, and Key Laboratory of Information Storage System, Ministry of Education of China. We are grateful to our shepherd, Andrea C. Arpaci-Dusseau, and the anonymous reviewers for their constructive comments and suggestions. We have released the source code for public use in GitHub.

References

[1] Intel® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/en-us/isaextensions, 2019.

[2] Intel® Optane$^{\text{TM}}$ DC persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[3] Intel® Threading Building Blocks. https://github.com/intel/tbb, 2019.

[4] Memcached. https://memcached.org/, 2019.

[5] Persistent Memory Development Kit. http://pmem.io/, 2019.

[6] pmemkv. http://pmem.io/pmemkv/index.html, 2019.

[7] Redis. https://redis.io/, 2019.

[8] The C++ Standard Template Library. http://www.cplusplus.com/reference/functional/hash/, 2020.

[9] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 11(5):553–565, 2018.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, United Kingdom, June 2012.

[11] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill Book Company, 1999.

[12] Daniel Bittman, Darrell D. E. Long, Peter Alvaro, and Ethan L. Miller. Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, February 2019.

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[14] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 8(7):786–797, 2015.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, USA, June 2010.

[16] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

[18] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, April 2013.

[19] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory Comput. Syst.*, 55(3):475–520, 2014.

[20] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. An Efficient Wait-free Resizable Hash Table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*, Vienna, Austria, July 2018.

[21] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *15th International Symposium on Distributed Computing (DISC '01)*, Lisbon, Portugal, October 2001.

[22] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, USA, February 2018.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[25] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and Practical Locking with Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[26] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, USA, February 2017.

[27] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[28] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Ninth Eurosys Conference 2014 (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[29] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, Winnipeg, Manitoba, Canada, August 2002.

[30] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, February 2019.

[31] Nhan Nguyen and Philippas Tsigas. Lock-Free Cuckoo Hashing. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14)*, Madrid, Spain, June 2014.

[32] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, San Francisco, CA, USA, June 2016.

[33] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA '03)*, Anaheim, California, USA, February 2003.

[34] Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo. Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, USA, July 2019.

[35] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering (ICDE '18)*, Paris, France, April 2018.

[36] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA, February 2020.

[37] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, USA, February 2015.

[38] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, USA, July 2019.

[39] Pengfei Zuo and Yu Hua. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Trans. Parallel Distrib. Syst.*, 29(5):985–998, 2018.

[40] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.

Optimizing Memory-mapped I/O for Fast Storage Devices

Anastasios Papagiannis[1], Giorgos Xanthakis[1], Giorgos Saloustros,
Manolis Marazakis, and Angelos Bilas[1]
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
{apapag, gxanth, gesalous, maraz, bilas}@ics.forth.gr

Abstract

Memory-mapped I/O provides several potential advantages over explicit read/write I/O, especially for low latency devices: (1) It does not require a system call, (2) it incurs almost zero overhead for data in memory (I/O cache hits), and (3) it removes copies between kernel and user space. However, the Linux *memory-mapped I/O* path suffers from several scalability limitations. We show that the performance of Linux *memory-mapped I/O* does not scale beyond 8 threads on a 32-core server. To overcome these limitations, we propose *FastMap*, an alternative design for the *memory-mapped I/O* path in Linux that provides scalable access to fast storage devices in multi-core servers, by reducing synchronization overhead in the common path. *FastMap* also increases device queue depth, an important factor to achieve peak device throughput. Our experimental analysis shows that *FastMap* scales up to 80 cores and provides up to 11.8× more IOPS compared to *mmap* using *null_blk*. Additionally, it provides up to 5.27× higher throughput using an Optane SSD. We also show that *FastMap* is able to saturate state-of-the-art fast storage devices when used by a large number of cores, where Linux *mmap* fails to scale.

1 Introduction

The emergence of fast storage devices, with latencies in the order of a few μs and IOPS rates in the order of millions per device is changing the I/O landscape. The ability of devices to cope well with random accesses leads to new designs for data storage and management that favor generating small and random I/Os to improve other system aspects [2, 35, 42, 43]. Although small and random I/Os create little additional pressure to the storage devices, they result in significantly higher CPU overhead in the kernel I/O path. As a result, the overhead of performing I/O operations to move data between memory and devices is becoming more pronounced, to the point where a large fraction of server CPU cycles are consumed only to serve storage devices [11, 46].

In this landscape, *memory-mapped I/O*, i.e. Linux *mmap*, is gaining more attention [8, 13, 24, 42, 43] for data intensive applications because of its potentially lower overhead compared to read/write system calls. An off-the-shelf NVMe block device [28] has access latency close to 10 μs and is capable of more than 500 KIOPS for reads and writes. Byte-addressable, persistent memory devices exhibit even better performance [29]. The traditional read/write system calls in the I/O path incur overheads of several μs [11, 46] in the best case and typically even higher, when asynchronous operations are involved.

In contrast, when using *memory-mapped I/O* a file is mapped to the process virtual address space where the user can access data with processor *load*/*store* instructions. The kernel is still responsible for moving data between devices and memory; *mmap* removes the need for an explicit system call per I/O request and incurs the overhead of an implicit page fault only when data does not reside in memory. In the case when data reside in memory, there is no additional overhead due to I/O cache lookups and system calls. Therefore, the overhead for hits is reduced dramatically as compared to both the kernel buffer cache but also to user-space I/O caches used in many applications. In several cases *memory-mapped I/O* removes the need to serialize and deserialize user data, by allowing applications to have the same format for both in-memory and persistent data, and also the need for memory copies between kernel and user space.

A major reason for the limited use of *memory-mapped I/O*, despite its advantages, has been that *mmap* may generate small and random I/Os. With modern storage devices, such as NVMe and persistent memory, this is becoming less of a concern. However, Figure 1 shows that the default *memory-mapped I/O* path (*mmap* backed by a device) for random page faults does not scale well with the number of cores. In this experiment (details in Section 4), we use *null_blk*, a Linux driver that emulates a block device but does not issue I/Os to a real device (we use 4TB dataset and 192GB of

[1] Also with the Department of Computer Science, University of Crete, Greece.

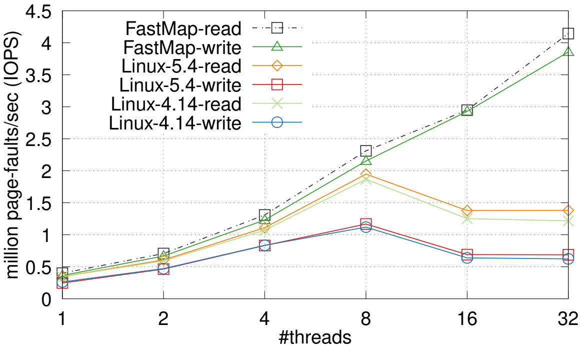


Figure 1: Scalability of random page faults using two versions of Linux *memory-mapped I/O* path (v4.14 & v5.4) and *FastMap*, over the *null_blk* device.

DRAM cache). Using *null_blk* allows us to stress the Linux kernel software stack while emulating a low-latency, next-generation storage device. Linux *mmap* scales up to only 8 cores, achieving 7.6 GB/s (2M random IOPS), which is about 5× less compared to a state-of-the-art device [29]; servers with multiple storage devices need to cope with significantly higher rates. We observe that from Linux kernel 4.14 to 5.4 the performance and the scalability of the *memory-mapped I/O* path has not improved significantly. Limited scalability also results in low device queue depth. Using the same micro-benchmark for random read page faults with 32 threads on an Intel Optane SSD DC P4800X, we see that the average device queue depth is 27.6. A large queue depth is essential for fast storage devices to provide their peak device throughput.

In this paper, we propose *FastMap*, a novel design for the *memory-mapped I/O* path that overcomes these two limitations of *mmap* for data intensive applications on multi-core servers with fast storage devices. *FastMap* (a) separates clean and dirty-trees to avoid all centralized contention points, (b) uses full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduces a scalable DRAM cache with per-core data structures to reduce latency variability. *FastMap* achieves both higher scalability and higher I/O concurrency by (1) avoiding all centralized contention points that limit scalability, (2) reducing the amount of CPU processing in the common path, and (3) using dedicated data-structures to minimize interference among processes, thus improving tail latency. As a further extension to *mmap*, we introduce a user-defined read-ahead parameter to proactively map pages in application address space and reduce the overhead of page faults for large sequential I/Os.

We evaluate *FastMap* using both micro-benchmarks and real workloads. We show that *FastMap* scales up to 80 cores and provides up to 11.8× more random IOPS compared to Linux *mmap* using *null_blk*. *FastMap* achieves 2× higher throughput on average for all YCSB workloads over Kreon [43], a persistent key-value store designed to use *memory-mapped I/O*. Moreover, we use *FastMap* to extend the virtual address space of memory intensive applications beyond the physical memory size over a fast storage device. We achieve up to 75× lower average latency for TPC-C over Silo [54] and 5.27× better performance with the Ligra graph processing framework [50]. Finally, we achieve 6.06% higher throughput on average for all TPC-H queries over MonetDB [8] that mostly issue sequential I/Os.

In summary, our work optimizes the *memory-mapped I/O* path in the Linux kernel with three main contributions:

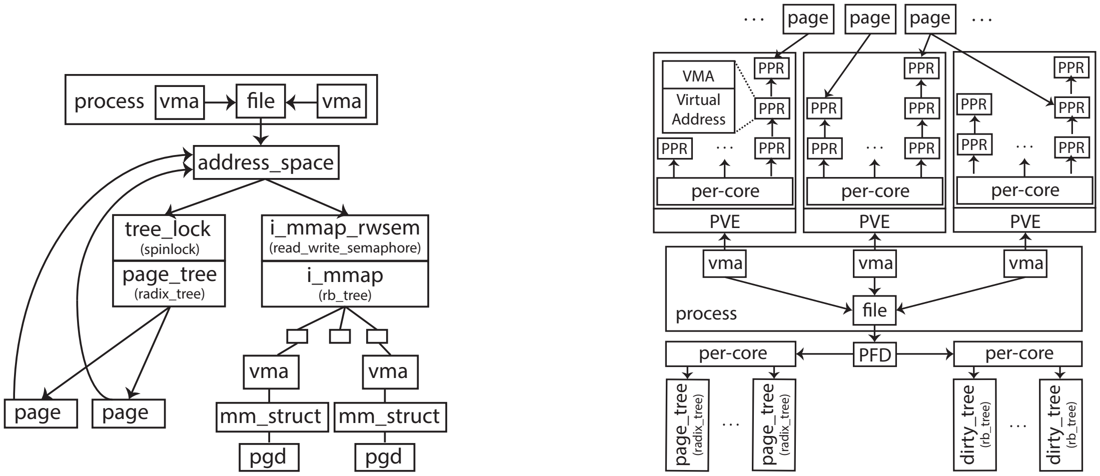
1. We identify severe performance bottlenecks of Linux *memory-mapped I/O* in multi-core servers with fast storage devices.
2. We propose *FastMap*, a new design for the *memory-mapped I/O* path.
3. We provide an experimental evaluation and analysis of *FastMap* compared to Linux *memory-mapped I/O* using both micro-benchmarks and real workloads.

The rest of the paper is organized as follows. §2 provides the motivation behind *FastMap*. §3 presents the design of *FastMap* along with our design decisions. §4 and §5 present our experimental methodology and results, respectively. Finally, §6 reviews related work and §7 concludes the paper.

2 Motivation

With storage devices that exhibit low performance for random I/Os, such as hard disk drives (HDDs), *mmap* results in small (4KB) random I/Os because of the small page size used in most systems today. In addition, *mmap* does not provide a way for users to manage page writebacks in the case of high memory pressure, which leads to unpredictable tail latencies [43]. Therefore, historically the main use of *mmap* has been to load binaries and shared libraries into the process address space; this use-case does not require frequent I/O, uses read-mostly mappings, and exhibits a large number of shared mappings across processes, e.g. libc is shared by almost all processes of the system. Reverse mappings provide all page table translations for a specific page and they are required in order to unmap a page during evictions. Therefore, Linux *mmap* uses object-based reverse mappings [37] to reduce memory consumption and enable fast *fork* system calls, as they do not require copying full reverse mappings.

With the introduction of fast storage devices, where the throughput gap between random and sequential I/O is small, *memory-mapped I/O* has the potential to reduce I/O path overhead in the kernel, which is becoming the main bottleneck for data-intensive applications. However, data intensive applications, such as databases or key-value stores, have different requirements compared to loading binaries: they can be write-intensive, do not require large amount of sharing, and do not use *fork* system calls frequently. These properties make the



(a) Linux

(b) *FastMap*

Figure 2: Linux (left) and *FastMap* (right) high-level architecture for memory-mapped files (acronyms: **PFD**=**P**er-**F**ile-**D**ata, **PVE**=**P**er-**V**ma-**E**ntry, **PPR**=**P**er-**P**ve-**R**map).

use of full reverse mappings a preferred approach. In addition, data intensive applications use datasets that do not fit in main memory and thus, the path of reading and writing a page from the device becomes the common case. Most of these applications are also heavily multithreaded and modern servers have a large number of cores.

3 Design of *FastMap*

The Linux kernel provides the *mmap* and *munmap* system calls to create and destroy memory mappings. Linux distinguishes memory mappings in shared vs. private. Mappings can also be anonymous, i.e. not backed by a file or device. Anonymous mappings are used for memory allocation. In this paper we examine I/O over persistent storage, an inherently shared resource. Therefore, we consider only shared memory mappings backed by a file or block device, as also required by Linux *memory-mapped I/O*.

Figure 2(a) shows the high-level architecture of shared memory mappings in the Linux kernel. Each virtual memory region is represented by a *struct vm_area_struct* (*VMA*). Each *VMA* points to a *struct file* (*file*) that represents the backing file or device and the starting offset of the memory mapping to it. Each *file* points to (a shared between processes) *struct address_space* (*address_space*) which contains information about mapped pages and the backing file or device.

Figure 2(b) illustrates the high-level design of *FastMap*. The most important components in our design are *per_file_data* (*PFD*) and *per_vma_entry* (*PVE*). Combined, these two components provide equivalent functionality as the Linux kernel *address_space* structure. Each *file* points to a *PFD* and each *VMA* points to a *PVE*. The role of a *PFD* is to keep metadata about device blocks that are in the *FastMap* cache and metadata about dirty pages. *PVE* provides full reverse mappings.

3.1 Separate Clean and Dirty Trees in *PFD*

In Linux, one of the most important parts of *address_space* is *page_tree*, a radix tree that keeps track of all pages of a cacheable and mappable file or device, both clean and dirty. This data structure provides an effective way to check if a device block is already in memory when a page fault occurs. Lookups are lock-free (*RCU*) but inserts and deletes require a spinlock (named *tree_lock*). Linux kernel radix trees also provide user-defined *tags* per entry. A *tag* is an integer, where multiple values can be stored using bitwise operations. In this case *tags* are used to mark pages as dirty. Marking a previously read-only page as writable requires holding the *tree_lock* to update the *tag*.

Using the experiments of Figure 1 and *lockstat* we see that *tree_lock* is by far the most contended lock: Using the same multithreaded benchmark as in Figure 1, over a single memory mapped region, *tree_lock* has 126× more contended lock acquisitions, which involve cross-cpu data, and 155× more time waiting to acquire the lock, compared to the second most contended lock. The second more contended lock is a spinlock that protects concurrent modifications in *PTEs* (4th level entries in the page table). This design has remained essentially unchanged from Linux kernel 2.6 up to 5.4 (latest stable version at the time of this writing).

To remove the bottleneck in *tree_lock*, *FastMap* uses a new structure for per-file data, *PFD*. The most important aspects of *PFD* are: (i) a per-core radix tree (*page_tree*) that keeps all (clean and dirty) pages and (ii) a per-core red-black tree (*dirty_tree*) that keeps only dirty pages. Each of these data structures is protected by a separate (per core) spinlock, different for the radix and red-black trees. We assign pages to cores in a round-robin manner and we use the page offset to identify the per-core structure that holds each page.

We use *page_tree* to provide lock-free lookups (RCU), similar to the Linux kernel. We use per-core data structures to reduce contention in case we need to add or remove a page. On the other hand, we do not use *tags* to mark pages as dirty but we use the *dirty_tree* for this purpose. In the case where we have to mark a previously read-only page as read-write, we only acquire the appropriate lock of *dirty_tree* without performing any additional operations to the *page_tree*. Furthermore, having all dirty pages in a sorted data structure (red-black tree) enables efficient I/O merging for the cases of writebacks and the *msync* system call.

3.2 Full Reverse Mappings in *PVE*

Reverse (inverted) mappings are also an important part of *mmap*. They are used in the case of evictions and writebacks and they provide a mechanism to find all existing virtual memory mappings of a physical page. File-backed memory mappings in Linux use object-based reverse mappings [37]. The main data structure for this purpose is a red-black tree, *i_mmap*. It contains all *VMAs* that map at least one page of this *address_space*. A read-write semaphore, *i_mmap_rwsem*, protects concurrent accesses to the *i_mmap* red-black tree. The main function that removes memory mappings for a specific page is *try_to_unmap*. Each page has two fields for this purpose: (i) a pointer to the *address_space* that it belongs to and (ii) an atomic counter (*_mapcount*) that keeps the number of active page mappings. Using the pointer to *address_space*, *try_to_unmap* gets access to *i_mmap* and then iterates over all *VMAs* that belong to this mapping. Through each VMA, it has access to *mm_struct* which contains the root of the process page table (*pgd*). It calculates the virtual address of the mapping based on the *VMA* and the page, which is required for traversing the page table. Then it has to check all active *VMAs* of *i_mmap* if the specific page is mapped, which results in many useless page table traversals. This is the purpose of *_mapcount*, which limits the number of traversals. This strategy is insufficient in some cases but it requires a very small amount of memory for the reverse mappings. More specifically, in the case where *_mapcount* is greater than zero, we may traverse the page table for a *VMA* where the requested page is not mapped. This can happen in the case where a page is mapped in several different *VMAs* in the same process, i.e. with multiple *mmap* calls, or mapped in the address space of multiple different processes. In such a case, we have unnecessary page table traversals that introduce overheads and consume CPU cycles. Furthermore, during this procedure, *i_mmap_rwsem* is held as a read lock and as a write lock only during *mmap* and *munmap* system calls. Previous research shows that even a read lock can limit scalability in multicore servers [15].

The current object-based reverse mappings in Linux have two disadvantages: (1) with high likelihood they result in unnecessary page table traversals, originating from *i_mmap*, and (2) they require a coarse grain read lock to iterate *i_mmap*. Other works have shown that in multi-core servers locks can be expensive, even for read-write locks when acquired as read locks [15]. These overheads are more pronounced in servers with a NUMA memory organization [10].

To overcome these issues *FastMap* provides finer grained locking, as follows: *FastMap* uses a structure with an entry for each VMA, *PVE*. Each *PVE* keeps a per-core list of all pages that belong to this *VMA*. A separate (per core) spinlock protects each of these lists. The lists are append-only as unmapping a page from a different page table does not require any ordering. We choose the appropriate list based on the core that runs the append operation (*smp_processor_id()*). These lists contain *per_pve_rmap* (*PPR*) entries. Each *PPR* contains a tuple $(VMA, virtual\_address)$. These metadata are sufficient to allow iterating over all mapped pages of a specific memory mapping in the case of an *munmap* operation. Furthermore, each page contains an append-only list of active *PPRs*, which are shared both for *PVEs* and pages. This list is used when we need to evict a page that is already mapped in one or more address spaces, in the event of memory pressure.

3.3 Dedicated DRAM Cache

An *mmap address_space* contains information about the backing file or device and the required functions to interact with the device in case of page reads and writes. To write back a set of pages of a memory mapping, Linux iterates *page_tree* in a lock-free manner with RCU and writes only the pages that have the dirty tag enabled. Linux also keeps a per-core LRU to find out which pages to evict. In the case of evictions, Linux tries to remove clean pages in order not to wait for dirty pages to do the writeback [37].

The Linux page-cache is tightly coupled with the swapper. For the *memory-mapped I/O* path, this dependency results in unpredictable evictions and bursty I/O to the storage devices [43]. Therefore, *FastMap* implements its own DRAM cache, managing evictions via an approximation of LRU. *FastMap* has two types of LRU lists: one containing only clean pages (*clean_queue*) and one containing only dirty pages (*dirty_queue*). *FastMap* maintains per-core *clean_queues* to reduce lock contention. We identify the appropriate *clean_queue* as clean_queue_id = page_offset % num_cores.

When there are no free pages during a page fault, *FastMap* evicts only clean pages, similar to the Linux kernel [37], from the corresponding *clean_queue*. We evict a batch (with a configurable size, currently set to 512) of clean pages to amortize the cost of page table manipulation and TLB invalidations. Each page eviction requires a TLB invalidation with the *flush_tlb* function, if the page mapping is cached. *flush_tlb* sends an IPI (Inter-Processor-Interrupt) to all cores, incurring significant overheads and limiting scalability [3,4]. We implement a mechanism to reduce the number of calls to *flush_tlb* function, using batching, as follows.

A TLB invalidation requires a pointer to the page table and the *page_offset*. *FastMap* keeps a pointer to the page table and a range of *page_offsets*. Then, we invoke *flush_tlb* for the whole range. This approach may invalidate more pages, but reduces the number of *flush_tlb* calls by a factor of the batch-size of page evictions (currently 512). As the file mappings are usually contiguous in the address space in data intensive applications, in the common case false TLB invalidations are infrequent. Thus, *FastMap* manages to maintain a high number of concurrent I/Os to devices and increase device throughput. *LATR* [33] proposes the use of an asynchronous TLB invalidation mechanism based on message passing. In our case, we cannot delay TLB invalidations as the pages should be used immediately for page fault handling.

FastMap uses multiple threads to write dirty pages to the underlying storage device (writeback). Each of these manages its own *dirty_queue*. This design removes the need of synchronization when we remove dirty pages from a *dirty_queue*. During writebacks, *FastMap* merges consecutive I/O requests to generate large I/O operations to the underlying device. To achieve this, we use *dirty_trees* that keep dirty pages sorted based on the device offset. As we have multiple *dirty_trees*, we initialize an iterator for each tree and we combine the iterator results using a min-max heap. When a writeback occurs, we also move the page to the appropriate *clean_queue* to make it available for eviction. As page writeback also requires a TLB invalidation, we use the same mechanism as in the eviction path to reduce the number of calls to the kernel *flush_tlb* function. Each writeback thread checks the ratio of dirty to clean pages and starts the writeback when the percentage is higher than 75% of the total cache pages. The cache in *FastMap* currently uses a static memory buffer, allocated upon module initialization and does not create any further memory pressure to the Linux page cache. We also provide a way to grow and shrink this cache at runtime, but we have not yet evaluated alternative sizing policies.

To keep track of free pages *FastMap* uses a per-core free list with a dedicated spinlock. During a major page fault i.e., when the page does not reside in the cache, the faulting thread first tries to get a page from its local free list. If this fails, it tries to steal a page from another core's free list (randomly selected). After *num_cores* unsuccessful tries, *FastMap* forces page evictions to cleanup some pages. To maintain all free lists balanced, each evicted page is added to the free list from which we originally obtained the page.

Overall, *FastMap* with per-core data structures requires more memory compared to the native Linux *mmap*. *FastMap* requires a single *PFD*, which is 1120 bytes, per file for all memory mappings. A single *PVE* is about 512 bytes and a single *PPR* is 24 bytes. We require a single *PVE* for each mmap call, i.e. 1 : 1 with the Linux VMA struct. *FastMap* requires a single *PPR* entry per *PVE* for each mapped page, independently of how many threads access the same page. In the setups we target, there is little sharing of files across processes and we can therefore, assume that we only need one *PPR* entry for each page in our DRAM cache. For instance, assume that a single application maps 1000 files and uses $8GB$ of DRAM cache. This results in $1.64MB$ of additional memory, independently of the size of files and the number of threads. *FastMap* targets storage servers with large memory spaces and can be applied selectively for the specific mount points that hold the files of data-intensive applications. While it is, in principle, possible to allow more fine-grain uses of *FastMap* in Linux, we leave this possibility for future work.

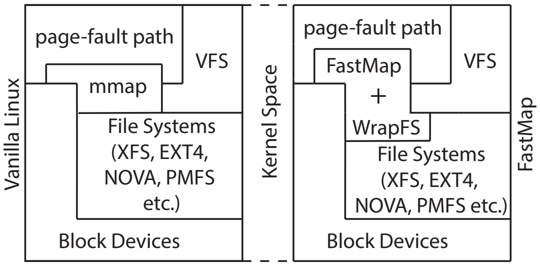


Figure 3: FastMap I/O path.

Finally, the Linux kernel also provides private, file-backed memory mappings. These are Copy-On-Write mappings and writes to them do not reach the underlying file/device. Such mappings are outside the scope of this paper, but they share the same path in the Linux kernel to a large extent. Our proposed techniques also apply to private file-backed mappings. However, these mappings are commonly used in Linux kernel to load binaries and shared libraries, resulting in a large degree of sharing. We believe that it is not beneficial to use the increased amount of memory required by *FastMap* to optimize this relatively uncommon path.

3.4 Implementation

Figure 3 shows the I/O path in the Linux kernel and indicates where *FastMap* is placed. *FastMap* is above *VFS* and thus is independent of the underlying file system. This means that common file systems such as *XFS*, *EXT4*, and *BTRFS* [1] can benefit from our work.

FastMap provides a user interface for accessing both a block device but also a file system through a user-defined mount point. For the block device case, we implement a virtual block device that uses our custom *mmap* function. All other block device requests (e.g. *read*/*write*) are forwarded to

[1] *FastMap* has been successfully tested with all of these file systems.

the underlying device. Requests for fetching or evicting pages from *FastMap* are issued directly to the underlying device.

For the file system implementation we use WrapFS [59], a stackable file system that intercepts all *mmap* calls to a specific mount point so that *FastMap* is used instead of the native Linux *mmap* implementation. For fetching or evicting pages from within *FastMap* we use direct I/O to the underlying file system, bypassing the Linux page cache. All other file system calls are forwarded to the underlying file system.

4 Experimental Methodology

In this section, we present the experimental methodology we use to answer the following questions:

1. How does *FastMap* perform compared to Linux *mmap*?
2. How much does *FastMap* improve storage I/O?
3. How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

Our main testbed consists of a dual-socket server that is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, each with 8 physical cores and 16 hyper-threads for a total of 32 hyper-threads. The primary storage device is a PCIe-attached Intel Optane SSD DC P4800X series [28] with 375 GB capacity. For the purposes of evaluating scalability, we use an additional four-socket server. This four-socket server is equipped with four Intel(R) Xeon(R) CPU E5-4610 v3 CPUs running at 1.7 GHz, each with 10 physical cores and 20 hyper-threads for a total of 80 hyper-threads. Both servers are equipped with 256 GB of DDR4 DRAM at 2400 MHz and run CentOS v7.3, with kernel 4.14.72.

During our evaluation we limit the available capacity of DRAM (using a kernel boot parameter) as required by different experiments. In our evaluation we use datasets that both fit and do not fit in main memory. This allows us to provide a more targeted evaluation and separate the costs of the page-fault path and the eviction path. To reduce variability in our experiments, we disable swap and Transparent Huge Pages (THP), and we set the CPU scaling governor to "performance". In experiments where we want to stress the software path of the Linux kernel we also use the *null_blk* [40] and *pmem* [47] block devices. *null_blk* emulates a block device but ignores the I/O requests issued to it. For *null_blk* we use the bio-based configuration. *pmem* emulates a fast block device that is backed by DRAM.

In our evaluation we first use a custom multithreaded microbenchmark. It uses a configurable number of threads that issue *load/store* instructions at randomly generated offsets within the memory mapped region. We ensure that each *load/store* results in a page fault.

Second, we use a persistent key-value store. We choose Kreon [43], a state-of-the-art persistent key-value store that is designed to work with *memory-mapped I/O*. The design of Kreon is similar to the LSM-tree, but it maintains a separate B-Tree index per-level to reduce I/O amplification. Kreon uses a log to keep user data. It uses *memory-mapped I/O* to perform all I/O between memory and (raw) devices. Furthermore, it uses Copy-On-Write (COW) for persistence, instead of Write-Ahead-Logging. Kreon follows a single-writer, multiple-reader concurrency model. Readers operate concurrently with writers using Lamport counters [34] per node for synchronization to ensure correctness. For inserts and updates, it uses a single lock per database; however, by using multiple databases Kreon can support concurrent updates.

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 1: Standard YCSB Workloads.

To improve single-database scalability in Kreon and make it more suitable for evaluating *FastMap*, we implement the second protocol that Bayer et al. propose [7]. This protocol requires a read-write lock per node. It acquires the lock as a read lock in every traversal from the root node to a leaf. In the case of inserts or rebalance operations it acquires the corresponding lock as a write lock. As every operation has to acquire the root node read lock, this limits scalability [15]. To overcome this limitation, we replace the read/write lock of the root node with a Lamport counter and a lock. Every operation that modifies the root node acquires the lock, changes the Lamport counter, performs a COW operation, and then writes the update in the COW node.

For Kreon we use the YCSB [18] workloads and more specifically a C++ implementation [48] to remove overheads caused by the JNI framework, as *Kreon* is highly efficient and is designed to take advantage of fast storage devices. Table 1 summarizes the YCSB workloads we use. These are the proposed workloads, and we execute them in the author's proposed sequence [18], as follows: LoadA, RunA, RunB, RunC, RunF, RunD, clear the database, and then LoadE, RunE.

Furthermore, we use Silo [54], an in-memory database that also provides scalable transactions for modern multicore machines. In this case, we use TPC-C [52], a transactional benchmark, which models a retail operation and is a common benchmark for OLTP workloads. We also use Ligra [50], a lightweight graph processing framework for shared memory with OpenMP-based parallelization. Specifically, we use the Breadth First Search (BFS) algorithm. We use Silo and Ligra to evaluate *FastMap*'s effectiveness in extending the virtual address space of an application beyond physical memory over fast storage devices. For this reason we convert all *malloc/free* calls of Ligra and Silo to allocate space over a memory-mapped file on a fast storage device. We use the *libvmmalloc* library from the Persistent Memory Development Kit (PMDK) [45]. *libvmmalloc* transparently converts all dy-

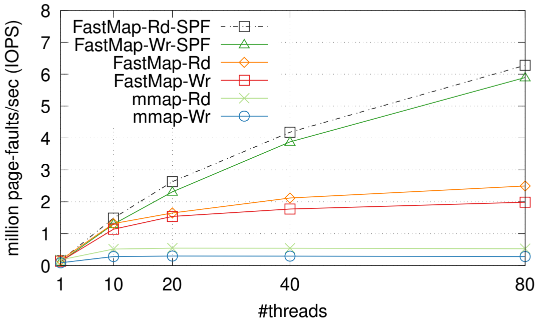


Figure 4: Scalability of random page faults for Linux and *FastMap*, with up to 80 threads, using the *null_blk* device.

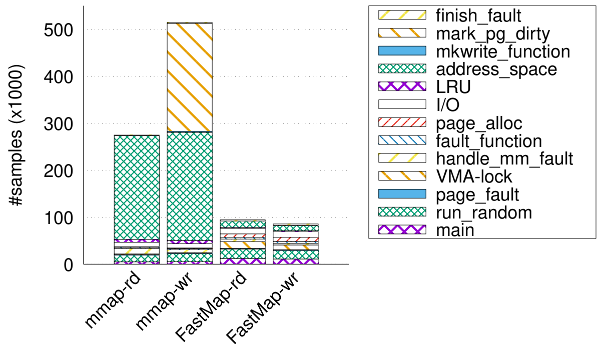


Figure 5: *FastMap* and Linux *mmap* breakdown for read and write page faults, with *null_blk* and 32 cores.

namic memory allocations to persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application. The memory allocator of *libvmmalloc* is based on *jemalloc* [30].

Finally, we evaluate *FastMap* using MonetDB-11.31.7 [8, 39], a column-oriented DBMS that is designed to use *mmap* to access files instead of using the read/write API. We use the TPC-H [53] benchmark, a warehouse read-mostly workload.

We run all experiments three times and we report the averages. In all cases the variation observed across runs is small.

5 Experimental Results

5.1 How does *FastMap* perform compared to Linux *mmap*?

Microbenchmarks: Figure 1 shows that Linux *mmap* fails to scale beyond eight threads on our 32-core server. *FastMap* provides 3.7× and 6.6× more random read and write IOPS, respectively, with 32 threads compared to Linux *mmap*. Furthermore, both versions 4.14 and 5.4 of the Linux kernel achieve similar performance. To further stress *FastMap*, we use our 80-core server and the *null_blk* device. Figure 4 shows that with 80 threads, *FastMap* provides 4.7× and 7× higher random read and write IOPS respectively, compared to Linux *mmap*. Furthermore, in both cases *FastMap* performs up to 38% better even in the case where there is little or no concurrency, when using a single thread.

Figure 4 shows that not only *FastMap* scales better compared to Linux *mmap*, but also that *FastMap* sustains more page faults per second. On the other hand *FastMap* does not achieve perfect scalability. For this reason, we profile *FastMap* using the random read page faults microbenchmark. We find that the bottleneck is the read-write lock (*mmap_sem*) that protects the red-black tree of active *VMAs*. This is the problem that *Bonsai* [15] tackles. Specifically, with 10 cores the cost of *read_lock* and *read_unlock* is 7.6% of the total execution time, with 20 cores it becomes 25.4%, with 40 cores 32%, and with 80 cores 37.4%. To confirm this intuition, we apply Speculative Page Faults (SPF) [20], and attempt to use SRCU (Sleepable RCU) instead of the read-write lock to protect the red-black tree of active *VMAs*, an approach similar to *Bonsai*. We use the Linux kernel patches from [19] as, at the time of writing, they have not been merged in the Linux mainline. As SPF works only for anonymous mappings, we modify it to use *FastMap* for block-device backed memory-mappings. Figure 4 shows that *FastMap* with *SPF* provides even better scalability: 2.51× and 11.89× higher read IOPS compared to *FastMap* without SPF and to Linux kernel, respectively. We do not provide an evaluation of *SPF* without *FastMap* as it (1) works only for anonymous mappings and (2) it could use the same Linux kernel path that has scalability bottlenecks, as we show in Section 3.1.

Figure 5 shows the breakdown of the execution time for both random reads and writes. We profile these runs using *perf* at 999Hz and plot the number of samples (y axis) that *perf* reports. First, we see that for random reads Linux *mmap* spends almost 80% of the time in manipulating the *address_space* structure, specifically in the contented *tree_lock* that protects the *radix_tree* which keeps all the pages of the mapping (see Section 3). In *FastMap* we do not observe a single high source of overhead. In the case of writes the overhead of this lock is even more pronounced in Linux *mmap*. For each page that is converted from read-only to read-write, Linux has to acquire this lock again to set the *tag*. *FastMap* removes this contention point as we keep metadata about dirty pages only in the per-core red-black trees (Section 3.3). Therefore, we do not modify the *radix_tree* upon the conversion of a read-only page to a read-write page.

Figure 6 shows how each optimization in *FastMap* affects I/O performance. Vanilla is the Linux *mmap* and *basic* is *FastMap* with all the optimizations disabled, except the per-core red-black tree. The *per-core radix-tree* optimization is important, because with increasing core counts on modern

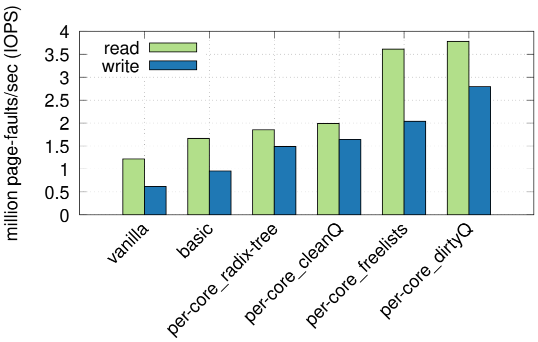


Figure 6: Performance gains from different optimizations in *FastMap*, as compared to "vanilla" Linux using *null_blk* and 32 cores.

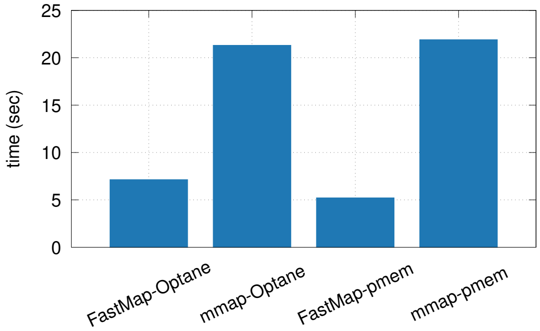


Figure 7: Execution time for Ligra running BFS with 32 threads and using an Optane SSD and a *pmem* device.

servers (Section 3.1) the single *radix tree* lock is by far the most contended lock. *per-core cleanQ* enables the per-core LRU list for clean pages. The *per-core freelists* optimization allows for scalable page allocation, resulting in significant performance gains. Finally, the main purpose of *per-core dirtyQ* is to enable higher concurrency when we convert a page from read-only to read-write and allow for multiple eviction threads with minimal synchronization. This optimization mainly improves the write path, as is shown in Figure 6.

In-memory Graph Processing: We evaluate *FastMap* as a mechanism to extend the virtual address space of an application beyond the physical memory and over fast storage devices. Using *mmap* (and *FastMap*) a user can easily map a file over fast storage and provide an extended address space, limited only by device capacity. We use Ligra [50], a graph processing framework, a demanding workload in terms of memory accesses and commonly operating on large datasets. Ligra assumes that the dataset (and metadata) fit in main memory. For our evaluation we generate a R-Mat [12] graph of 100M vertices, with the number of directed edges is set to 10× the number of vertices. We run BFS on the resulting 18GB graph, thus generating a read-mostly random I/O pattern. Ligra requires about 64GB of DRAM throughout execution. To evaluate *FastMap* and Linux *mmap*, we limit the main memory of our 32-core server to 8 GB and we use the Optane SSD device.

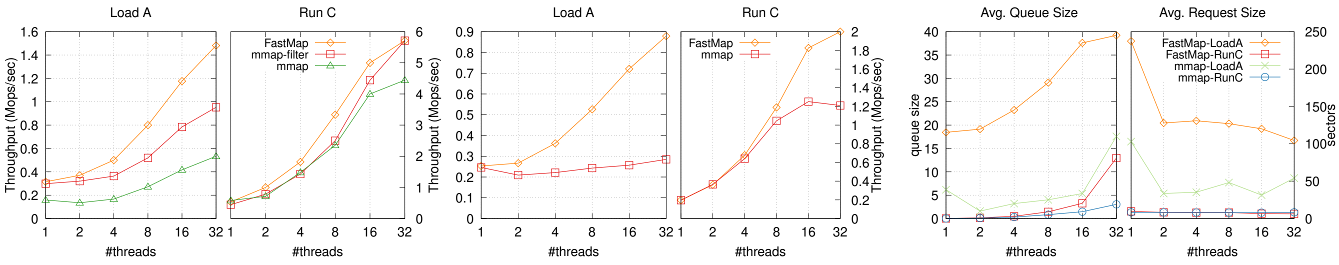
Figure 7 shows that BFS completes in 6.42s with *FastMap* compared to 21.3s with default *mmap* and achieves a 3.31× improvement. *FastMap* requires less than half the system time (10.3% for *FastMap* vs. 27.38% for Linux) and stresses more the underlying storage device as seen in iowait time (19.31% for *FastMap* vs. 9.5% for Linux). This leaves 2.11× more user-time available for the Ligra workload execution. Using a *pmem* device the benefits of *FastMap* are even higher. Linux *mmap* requires 21.9s for BFS, while *FastMap* requires only 4.15s, i.e. a 5.27× improvement. Overall, Ligra induces a highly concurrent I/O pattern that stresses the default *mmap*, resulting in lock contention as described in Section 3.1 and as evidenced by the increased system time. The default *mmap* results in a substantial slowdown, even with a *pmem* device that has throughput comparable to DRAM.

5.2 How much does *FastMap* improve storage I/O?

Kreon Persistent Key-value Store: In this section we evaluate *FastMap* using Kreon, a persistent key-value store that uses *memory-mapped I/O* and a dataset of 80M records. The keys are 30 bytes long, with 1000 byte values. This results in a total footprint of about 76GB. We issue 80M operations for each of the YCSB workloads. For the in-memory experiment, we use the entire DRAM space (256GB) of the testbed, whereas for the out-of-memory experiment we limit available memory to 16GB. In all cases we use the Optane SSD device.

Figure 8(a) illustrates the scalability of Kreon, using *FastMap*, Linux *mmap*, and *mmap-filter*, with a dataset that fits in main memory. The *mmap-filter* configuration is the default Linux *mmap* implementation augmented with a custom kernel module we have created to remove the unnecessary read I/O from the block device for newly allocated pages. Using 32 threads (on the 32-core server), *FastMap* achieves 1.55× and 2.77× higher throughput compared to *mmap-filter* and *mmap* respectively, using the LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 9% and 28% higher throughput compared to *mmap-filter* and *mmap* respectively. As we see *mmap-filter* performs always better, therefore, for the rest of the Kreon evaluation we use this configuration as our baseline.

Figure 8(b) shows the scalability of Kreon with *FastMap* and *mmap-filter* (denoted as *mmap*) using a dataset that does not fit in main memory. Using 32 threads (on the 32-core



(a) In-memory

(b) Out-of-memory

(c) Average queue & request size

Figure 8: Kreon scalability with increasing the number of threads (*(a)* and *(b)*). Average queue size and average request size for an out-of-memory experiment *(c)*. In all cases we use the Optane SSD.

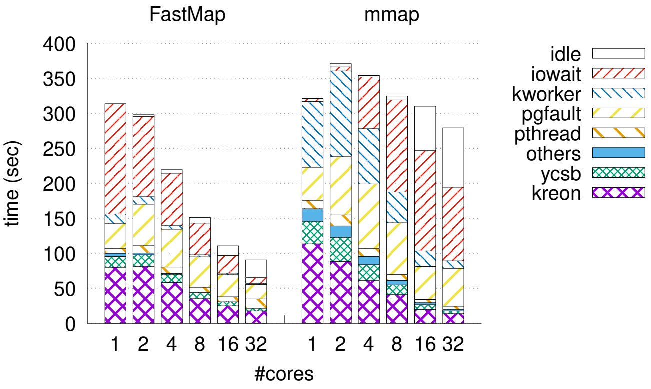


Figure 9: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment for LoadA YCSB workload, with an increasing number of cores, an equal number of YCSB threads, and the Optane SSD.

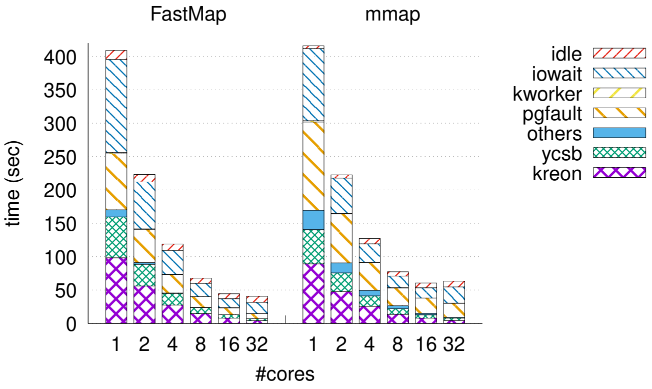


Figure 10: Kreon breakdown using *FastMap* and Linux *mmap* for an out-of-memory experiment with the RunC YCSB workload, with increasing number of cores (and equal number of YCSB threads) and the Optane SSD.

server) *FastMap* achieves 3.08× higher throughput compared to *mmap* using LoadA (insert only) workload. Using the RunC (read only) workload, *FastMap* achieves 1.65× higher throughput compared to *mmap*. We see that even for the lower core counts, *FastMap* outperforms *mmap* significantly. Next, we provide an analysis on what affects scalability in *mmap* and how *FastMap* behaves with an increasing number of cores.

Figure 9 shows the execution time breakdown for the out-of-memory experiment with an increasing number of threads for LoadA. *kworker* denotes the time spent in the eviction threads both for Linux *mmap* and *FastMap*. *pthread* refers to pthread locks, both mutexes and read-write locks as described in Section 4. First, we observe here that in the case of Linux *mmap* both *iowait* and *idle* time increases. For *iowait* time, the small queue depth that *mmap* generates (discussed in detail later) leads to sub-optimal utilization of the storage device. Furthermore, the idle time comes from sleeping in mutexes in the Linux kernel. We also observe that the *pgfault* time is lower in *FastMap* and this is more pronounced with 32 threads. In summary, the optimized page-fault path results in 2.64× lower *pgfault* time and 12.3× lower *iowait* time due to higher concurrency and larger average request size. In addition, the optimized page-fault path results in 3.39× lower idle time due to spinning instead of sleeping in the common path. This is made possible as we apply per-core locks to protect our data structures, which are less contended in the common case. Similar to the previous figure, Figure 10 shows the same metrics for RunC. In this case the breakdown is similar both for *FastMap* and Linux *mmap*. With 32 threads the notable differences are in *pgfault* and *iowait*. Linux *mmap* spends 2.88× and 1.41× more time for *pgfault* and *iowait*, respectively. The difference in *pgfault* comes from our scalable design for the *memory-mapped I/O* path. As in this case both systems always issue 4KB requests (page size), the difference in *iowait* comes from the higher queue depth achieved on average by *FastMap*.

Figure 8(c) shows the average queue depth and average request size for both *FastMap* and Linux *mmap*. Using 32 threads, *FastMap* produces higher queue depths for both LoadA and RunC, which is an essential aspect for high throughput with fast storage devices. With 32 threads in LoadA *FastMap* results in an average queue size of 39.2, while Linux *mmap* results in an average queue size of 17.5. Furthermore, *FastMap* also achieves a larger request size of 100.2 sectors (51.2KB) compared to 51.8 sectors (26.5KB) for Linux *mmap*. For RunC, the average request size is 8 sectors (4KB) for both *FastMap* and Linux *mmap*. However, *FastMap* achieves (with 32 threads) an average queue size of 13 compared to 3 for Linux *mmap*.

For all YCSB workloads, Kreon with *FastMap* outperforms Linux *mmap* by 2.48× on average (between $1.25-3.65\times$).

MonetDB: In this section we use TPC-H over MonetDB, a column oriented DBMS that uses *memory-mapped I/O* instead of *read*/*write* system calls. We focus on out-of-memory workloads, using a TPC-H dataset with a scale factor $SF = 50$ (around 50GB in size). We limit available server memory to 16GB and we use the Optane SSD device. In all 22 queries of TPC-H, system-time is below 10%. The use of *FastMap* further decreases the system time (between 5.4% and 48.6%) leaving more CPU cycles for user-space processing. In all queries, the improvement on average is 6.06% (between -7.2% and 45.7%). There are 4 queries where we have a small decrease in performance. Using profiling we see that this comes from the *map_pages* function that is responsible for the fault-around page mappings, and which is not as optimized in the current prototype. In some cases we see greater performance improvements compared to the reduction in system time. This comes from higher concurrency to the devices (I/O depth) which also results in higher read throughput. Overall, our experiments with MonetDB show that a complex real-life memory-based DBMS can benefit from *FastMap*. The queries produce a sequential access pattern to the underlying files which shows the effectiveness of *FastMap* also for this case.

5.3 How sensitive is *FastMap* to (a) file system choice and (b) false TLB invalidations?

In this section we show how underlying file system affects *FastMap* performance. Furthermore, we also evaluate the impact of batched TLB invalidations. For these purposes we use Silo [54], an in-memory key-value store that provides scalable transactions for multicores. We modify Silo to use a memory-mapped heap over both *mmap* and *FastMap*.

Table 2: Throughput and average latency for TPC-C.

	xput (kops/sec)	latency (ms)
mmap-EXT4-Optane SSD	4.3	7.43
mmap-EXT4-pmem	4.2	7.62
FastMap-*EXT4-Optane SSD*	226	0.141
FastMap-*EXT4-pmem*	319	0.101
FastMap-*NOVA-pmem*	344	0.009

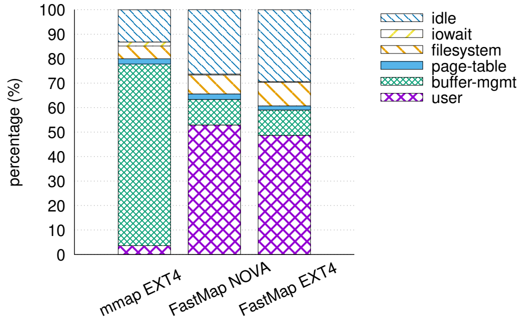


Figure 11: Execution time breakdown for Silo running TPC-C using different file systems and the *pmem* device.

File system choice: Table 2 shows the throughput and average latency of TPC-C over Silo. We use both EXT4 and NOVA. We also use XFS and BTRFS but we do not include these as they exhibit lower performance. We see that *FastMap* with EXT4 provides 52.5× and 75.9× higher throughput using an NVMe and a pmem device respectively, compared to *mmap*. We also see similar improvement in the average latency of TPC-C queries. With NOVA and a pmem device, *FastMap* achieves 1.07× higher throughput compared to EXT4. In all cases we do not use DAX *mmap*, as we have to provide DRAM caching over the persistent device. Therefore, *FastMap* improves performance of memory-mapped files over all file systems, although the choice of a specific file system does affect performance. In this case we see even larger performance improvements compared to Ligra and Kreon. Silo requires more page faults and it accesses a smaller portion of each page. Therefore, Silo is closer to a scenario with a single large file/device and a large number of threads generating page faults at random offsets. Consequently, it exhibits more of the issues we identify with Linux *mmap* compared to the other benchmarks: Kreon performs mostly sequential I/O for writes and a large part of a page is indeed needed when we do reads. From our evaluation we see that Ligra has better spatial locality compared to Silo and this explains the larger improvements we observe in Silo.

Figure 11 shows the breakdown of execution time for the previous experiments. In the case of Linux *mmap* with EXT4, most of the system time goes to buffer management: allocation of pages, LRUs, evictions, etc. In *FastMap*, this percentage is reduced from 74.2% to 10.3%, for both NOVA and EXT4. This results in more user-time available to TPC-C and increased performance. Finally, NOVA reduces system time compared to EXT4 and results in the best performance for TPC-C over Silo.

False TLB invalidations: *FastMap* uses batched TLB invalidations to provide better scalability and thus increased performance. Our approach reduces the number of calls to *flush_tlb_mm_range()*. This function uses Interprocessor Interrupts (IPI) to invalidate TLB entries in all cores and can result in scalability bottlenecks [3, 4, 16]. Batching of TLB invalidations can potentially result in increased TLB misses. In TPC-C over Silo, batching for TLB invalidations results in 25.5% more TLB misses (22.6% more load and 50.5% more store TLB misses). On the other hand, we have 24% higher throughput (ops/s) and 23.8% lower latency (ms). Using profiling, we see that without batching of TLB invalidations the system time spent in *flush_tlb_mm_range()* increases from 0.1% to 20.3%. We choose to increase the number of TLB misses in order to provide better scalability and performance. Other works [3,4,16] present alternative techniques to provide scalable TLB shootdown without increasing the number of TLB invalidations and can be potentially applied in *FastMap* for further performance improvements.

6 Related Work

We categorize related work in three areas: (a) replacing *read*/*write* system calls with *mmap* for persistence, (b) providing scalable address spaces, and (c) extending virtual address spaces beyond physical memory limits.

Using *memory-mapped I/O* in data-intensive applications: Both MonetDB [8] and MongoDB [13] (with *MMAP_v1* storage engine) are popular databases that use *mmap* to access data. When data fits in memory, *mmap* performs very well. It allows the application to access data at memory speed and removes the need for user-space cache lookups. Facebook's RocksDB [24], a persistent key-value store, provides both *read*/*write* and *mmap* APIs to access files. The developers of RocksDB state [26] that using *mmap* for an *in-memory* database with a *read-intensive* workload increases performance. However, they also state [25] that *mmap* sometimes causes problems when data does not fit in memory and is managed by a file system over a block device.

Tucana [42] and Kreon [43] are write-optimized persistent key-value stores that are designed to use *memory-mapped I/O* for persistence. The authors in [42] show that for a write-intensive workload the *memory-mapped I/O* results in excessive and unpredictable traffic to the devices, which results in freezes and increases tail-latency. Kreon [43] provides a custom *memory-mapped I/O* path inside the Linux kernel that improves write-intensive workloads and reduces the latency variability of Linux *mmap*. In this work, we address scalability issues and also present results for *memory-mapped I/O* with workloads beyond key-value stores.

DI-MMAP [22, 23], removes the swapper from the critical path and implements a custom (FIFO based) eviction policy using a fixed-size memory buffer for all *mmap* calls. This approach provides significant improvement compared to the default Linux *mmap* for HPC applications. We evaluate *FastMap* using more data-intensive applications, representative of data analytics and data serving workloads. In particular, our work assumes higher levels of I/O concurrency, and addresses scalability concerns with higher core counts.

FlashMap [27] combines memory (page tables), storage (file system), and device-level (FTL) indirections and checks in a single layer. *FastMap* provides specific optimizations only for the memory level and results in significant improvements in a file system and device agnostic manner.

2B-SSD [6] leverages SSD internal DRAM and the byte addressability of the PCIe interconnect to provide a dual, byte and block-addressable SSD device. It provides optimized write-ahead logging (WAL) over 2B-SSD for popular databases and results in significant improvements. FlatFlash [1] moves this approach further and provides a unified memory-storage hierarchy that results in even larger performance improvements. Both of these works move a large part of their design inside the device. *FastMap* works in a device-agnostic manner and provides specific optimizations in the operating system layer.

UMap [44] is a user-space memory-mapped I/O framework which adapts different policies to application characteristics and storage features. Handling page faults in user-space (using *userfaultfd* [31]) introduces additional overheads that are not acceptable in the case of fast storage devices. On the other hand, techniques proposed by *FastMap* can also be used in user-space memory-mapped I/O frameworks and provide better scalability in the page-fault path.

Similar to [14], *FastMap* introduces a read-ahead mechanism to amortize the cost of pre-faulting and improve sequential I/O accesses. However, our main focus is to reduce synchronization overheads in the common *memory-mapped I/O* path and enhance scalability on multicore servers. A scalable I/O path enables us to maintain high device queue depth, an essential property for efficient use of fast storage devices.

Byte-addressable persistent memory DIMMs, attached in memory slots, can be accessed similarly to DRAM with the processor *load*/*store* instructions. Linux provides Direct Access (DAX), a mechanism that supports direct mapping of persistent memory to user address space. File systems that provide a DAX mmap [17, 21, 56–58] bypass I/O caching in DRAM. On the other hand, other works [29] have shown that DRAM caching benefits applications when the working set fits in DRAM and can hide higher persistent memory latency compared to DRAM (by up to $\sim 3\times$). Accordingly, *FastMap* uses DRAM caching and supports both block-based flash storage and byte-addressable persistent memory. *FastMap* will benefit all DAX mmap file systems that need to provide

DRAM caching for *memory-mapped I/O*, as *FastMap* is file system agnostic.

Providing a scalable virtual address space: *Bonsai* [15] shows that anonymous memory mappings, i.e. not backed by a file or device, suffer from scalability issues. This type of memory mapping is mainly used for user memory allocations, e.g. *malloc*. The scalability bottleneck in this case is due to a contended read-write lock, named *mmap_sem*, that protects access to a red-black tree that keeps VMAs (valid virtual address spaces ranges). In the case of page faults, this lock is acquired as read lock. In the case of *mmap*/*munmap* this lock is acquired as write lock. Even in the read lock case, NUMA traffic in multicores limits scalability. *Bonsai* proposes the use of *RCU*-based binary tree for lock-free lookups, resulting in a system scaling up to 80 cores. *Bonsai* removes the bottleneck from concurrent page faults, but still serializes *mmap*/*munmap* operations even in non-overlapping address ranges.

In Linux, shared mappings backed by a file or device have a different path in the kernel, thus requiring a different design to achieve scalability. There are other locks (see Section 3.1) that cause scalability issues and *mmap_sem* does not result in any performance degradation. As we see from our evaluation of *FastMap*, using 80 cores the time spent in *mmap_sem* is 37.4% of the total execution time; therefore, *Bonsai* is complementary to our work and will also benefit our approach. Furthermore, authors in [32] propose an alternative approach to provide scalable address space operations, by introducing scalable range locks to accelerate non-conflicting virtual address space operations in Linux.

RadixVM [16] addresses the problem of serialization of *mmap*/*munmap* in non-overlapping address space ranges. This work is done in the *SV6* kernel and can also benefit from *FastMap* in a similar way to *Bonsai*.

The authors in [9] propose techniques to scale Linux for a set of kernel-intensive applications, but do not tackle the scalability limitations of *memory-mapped I/O*. In *pedsort* authors modify the application to use one process per core for concurrency and avoid the contention over the shared address space. In this paper we solve this issue at the kernel level, thus providing benefits to all user applications.

Extending the virtual address space over storage: The authors in [51] claim that by using *mmap* a user can effectively extend the main memory with fast storage devices. They propose a page reclamation procedure with a new page recycling method to reduce context switches. This makes it possible to use extended vector I/O – a parallel page I/O method. In our work, we implement a custom per-core mechanism for managing free pages. We also preallocate a memory pool to remove the performance bottlenecks identified in [51]. Additionally, we address scalability issues with *memory-mapped I/O*, whereas the work in [51] examines setups with up to 8 cores, where Linux kernel scales well.

FlashVM [49] uses a dedicated flash device for swapping virtual memory pages and provides flash-specific optimizations for this purpose. SSDAlloc [5] implements a hybrid DRAM/flash memory manager and a runtime library that allows applications to use flash for memory allocations in a transparent manner. SSDAlloc proposes the use of $16-32\times$ more flash than DRAM compared to FlashVM and to handle this increase they introduce a log-structured object store. Instead, *FastMap* targets the storage I/O path and reduces the overhead of *memory-mapped I/O*. *FastMap* is not a replacement for swap nor does it provide specific optimizations to extend the process address space over SSDs.

NVMalloc [55] enables client applications in supercomputers to allocate and manipulate memory regions from a distributed block-addressable SSD store (over FUSE [36]). It exploits the *memory-mapped I/O* interface to access local or remote NVM resources in a seamless fashion for volatile memory allocations. NVMalloc uses Linux *mmap*. Consequently, it can also benefit from *FastMap* at large thread counts combined with fast storage devices.

SSD-Assisted Hybrid Memory [41] augments DRAM with SSD storage as an efficient cache in object granularity for Memcached [38]. Authors claim that managing a cache at a page granularity incurs significant overhead. In our work, we provide an application agnostic approach at page granularity and we optimize scalability in the common path.

7 Conclusions

In this paper we propose *FastMap*, an optimized *memory-mapped I/O* path in the Linux kernel that provides a low-overhead and scalable way to access fast storage devices in multi-core servers. Our design enables high device concurrency, which is essential for achieving high throughput in modern servers. We show that *FastMap* scales up to 80 cores and provides up to $11.8\times$ more random IOPS compared to *mmap*. Overall, *FastMap* addresses important limitations of Linux *mmap* and makes it appropriate for data-intensive applications in multi-core servers over fast storage devices.

Acknowledgements

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the projects EVOLVE (Grant Agreement ID: 825061) and ExaNeSt (Grant Agreement ID: 671553). Anastasios Papagiannis is also supported by the Facebook Graduate Fellowship. Finally, we thank the anonymous reviewers for their insightful comments and our shepherd Sudarsun Kannan for his help with preparing the final version of the paper.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[3] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, Santa Clara, CA, July 2017. USENIX Association.

[4] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 211–224, USA, 2011. USENIX Association.

[6] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 425–438. IEEE Press, 2018.

[7] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[8] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, December 2008.

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 1–16, USA, 2010. USENIX Association.

[10] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[13] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[14] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[15] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[16] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[17] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud

serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] Laurent Dufour. Speculative page faults (Linux 4.14 patch). https://lkml.org/lkml/2017/10/9/180, 2017.

[20] Laurent Dufour. Speculative page faults. https://lwn.net/Articles/786105/, 2019.

[21] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[22] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 731–735, Nov 2012.

[23] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap–a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, March 2015.

[24] Facebook. RocksDB. https://rocksdb.org/. Accessed: June 4, 2020.

[25] Facebook. RocksDB IO. https://github.com/facebook/rocksdb/wiki/IO. Accessed: June 4, 2020.

[26] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed: June 4, 2020.

[27] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 580–591, New York, NY, USA, 2015. Association for Computing Machinery.

[28] INTEL. OPTANE SSD DC P4800X SERIES. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html. Accessed: June 4, 2020.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[30] jemalloc. http://jemalloc.net/. Accessed: June 4, 2020.

[31] Linux kernel. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt. Accessed: June 4, 2020.

[32] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 651–664, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Linux FUSE (Filesystem in Userspace). https://github.com/libfuse/libfuse. Accessed: June 4, 2020.

[37] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox professional guides. Wiley, 2008.

[38] Memcached. https://memcached.org/. Accessed: June 4, 2020.

[39] MonetDB. https://www.monetdb.org/Home. Accessed: June 4, 2020.

[40] Null block device driver. https://www.kernel.org/doc/Documentation/block/null_blk.txt. Accessed: June 4, 2020.

[41] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. In *2012 41st International Conference on Parallel Processing*, pages 470–479, Sep. 2012.

[42] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[43] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 490–502, New York, NY, USA, 2018. ACM.

[44] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78, Nov 2019.

[45] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/. Accessed: June 4, 2020.

[46] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[47] pmem.io: Persistent Memory Programming. http://pmem.io/. Accessed: June 4, 2020.

[48] Jinglei Ren. YCSB-C. https://github.com/basicthinker/YCSB-C, 2016.

[49] Mohit Saxena and Michael M. Swift. Flashvm: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 14, USA, 2010. USENIX Association.

[50] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[51] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4):19:1–19:27, May 2016.

[52] TPC-C. http://www.tpc.org/tpcc/. Accessed: June 4, 2020.

[53] TPC-H. http://www.tpc.org/tpch/. Accessed: June 4, 2020.

[54] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[55] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 957–968, May 2012.

[56] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory and its extensions. *ACM Trans. Storage*, 9(3), August 2013.

[57] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[58] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 5–5, Berkeley, CA, USA, 1999. USENIX Association.

A Comprehensive Analysis of Superpage Management Mechanisms and Policies

Weixi Zhu, Alan L. Cox and Scott Rixner
Rice University
{wxzhu, alc, rixner}@rice.edu

Abstract

Superpages (2MB pages) can reduce the address translation overhead for large-memory workloads in modern computer systems. This paper clearly outlines the sequence of events in the life of a superpage and explores the design space of when and how to trigger and respond to those events. This provides a framework that enables better understanding of superpage management and the trade-offs involved in different design decisions. Under this framework, this paper discusses why state-of-the-art designs exhibit different performance characteristics in terms of runtime, latency and memory consumption. This paper illuminates the root causes of latency spikes and memory bloat and introduces Quicksilver, a novel superpage management design that addresses these issues while maintaining address translation performance.

1 Introduction

The physical memory size of modern computers continues to grow at a rapid pace. Furthermore, there is an ever expanding class of "large-memory" data-oriented applications — including databases, data analysis tools, and scientific computations — that can productively utilize all of this memory. While some of these applications expect the entirety of their data to reside within physical memory, others process data at a scale that far exceeds its size. These others either use out-of-core computation frameworks or implement schemes for caching data from secondary storage that avoid swapping by the virtual memory system. In either case, these applications have large memory footprints, so the cost of virtual-to-physical address translation significantly impacts their performance.

The use of *superpages*, or "huge pages", can reduce the cost of virtual-to-physical address translation. For example, the x86-64 architecture supports 2MB superpages. Using these superpages (1) eliminates one level from the hierarchical page table, thereby reducing the expected number of memory accesses to resolve a TLB miss, and (2) enables more efficient use of the TLB's limited number of entries. Intel's recent processors can store up to 1536 4KB or 2MB page mappings in their TLBs. Superpages can therefore increase these TLBs' coverage from around 6MB (0.009% of the physical memory in a computer with 64GB of DRAM) to 3GB. While this is still a small fraction of the computer's physical memory, it is far more likely to capture an application's short-term working set. The benefits of this increased coverage are obvious. The challenge, however, is for the operating system (OS) to transparently manage superpages in an effective manner.

This paper first defines the five distinct events in the life cycle of a transparently managed superpage, and then it analyzes the various state-of-the-art approaches to handling each event. Briefly, the events are as follows. First, a physical superpage must be allocated. Throughout this paper, unless stated otherwise, "superpage" refers to a 2MB page, so this is the act of acquiring a contiguous, aligned 2MB region from the physical memory allocator. Second, the physical superpage must be prepared. For anonymous memory, the entire 2MB region must be zeroed. For file-backed memory, the entire 2MB region must be read from secondary storage. Third, a superpage mapping from a 2MB aligned virtual memory region to the physical superpage must be created. Fourth, the mapping must be destroyed. Finally, the physical memory must be deallocated. FreeBSD, Linux's Transparent Huge Pages (THP), and two recently proposed systems (Ingens [20] and HawkEye [24]) differ in when these events are triggered (for instance, these events can be independent, grouped, synchronous, asynchronous, etc.) and the granularity of the operations (for instance, some operations can be performed incrementally or all at once). This classification of the events enables a more principled comparison of the policies, behaviors, and performance of these different systems.

This paper also presents several new observations about transparent superpage management. First, coupling physical allocation, preparation, and mapping of superpages, as is done in Linux's THP, leads to memory bloat and fewer superpage mappings. Second, while alleviating tail latency problems in server workloads, state-of-the-art asynchronous, "out-of-place" promotion delays physical superpage allocation and reduces address translation benefits. Third, speculatively allocating physical superpages enables "in-place" promotion and obviates the need for asynchronous, out-of-place promotion. Fourth, in combination, reserving physical superpages and delaying partial deallocation of those superpages as long as possible fights fragmentation, leading to more superpage usage and address translation benefits. Finally, bulk zeroing is more efficient on modern processors than repeated 4KB zeroing. These observations are supported by evidence presented throughout the paper.

Finally, this paper introduces Quicksilver[1], an innovative transparent superpage management system that is based on FreeBSD's reservation-based physical memory allocator. Quicksilver achieves the benefits of aggressive superpage allocation, but mitigates the memory bloat and fragmentation issues that arise from underutilized superpages. Quicksilver is able to match or beat the performance of existing systems in scenarios with either lightly or heavily fragmented memory. For example, when using synchronous preparation, on a heavily fragmented system it achieves a 2x speedup over Linux for GraphChi performing PageRank on a dataset that exceeds the physical memory size. Furthermore, on Redis, Quicksilver is able to maintain the same throughput and tail latency as fragmentation increases, whereas the throughput of other systems degrades and tail latency increases. Finally, Quicksilver is able to limit memory bloat as well as Ingens [20], which is a recent research prototype specifically designed to combat memory bloat.

2 Transparent Superpage Management

Managing superpages transparently to the application involves five distinct events: physical superpage allocation, physical superpage preparation, superpage mapping creation, superpage mapping destruction, and physical superpage deallocation. Figure 1 illustrates the life cycle of a superpage in terms of these events. This section discusses the trade-offs between the possible choices, including those made by production and prototype systems [5, 16, 20, 23, 24], for when to trigger and how to handle these events.

2.1 Physical Superpage Allocation

The OS can choose to allocate a physical superpage to back any 2MB-aligned virtual memory region. A physical superpage could be allocated synchronously upon a page fault, or asynchronously via a background task. If there are free physical superpages, synchronous allocation is a relatively inexpensive operation given the widespread use of buddy allocators for physical memory management.

However, in order to allocate a physical superpage, the physical memory allocator must have an available, aligned, 2MB region. Under severe memory fragmentation, such regions may not be available. A memory manager could attempt to keep as many such regions available as possible (or create them when needed) using smart allocation policies or memory migration. If no such region is available or can be created, then the system must fall back to allocating 4KB pages.

Even after 4KB pages have been allocated for a virtual memory region, it is still possible to allocate a physical superpage for that region asynchronously. In the background, the OS can use migration to create free physical superpages or wait for them to be freed by applications. Once a free physical superpage exists, it could be allocated to a previously accessed virtual memory region. At that point, all previously allocated 4KB pages would need to be migrated into the newly acquired physical superpage.

2.2 Physical Superpage Preparation

Once a physical superpage has been allocated, it must be prepared with its initial data before it can be mapped. A physical superpage can be prepared in one of three ways. First, if the virtual memory region is anonymous, i.e., not backed by a file, then the page simply needs to be zeroed. Second, if the virtual memory region is a memory-mapped file, then the data must be read from the file. Finally, if the virtual memory region is currently mapped to 4KB pages, then the contents of those existing pages must be copied into the physical superpage. Note that any constituent pages that were not already mapped would need to be prepared appropriately, either via zeroing or reading from the backing file.

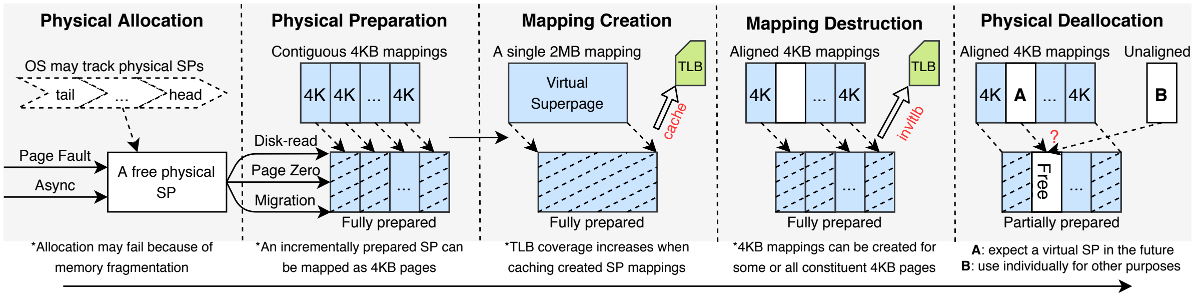
Physical superpages can be prepared all at once or incrementally. Furthermore, as they are prepared, they can have some, or all, of their constituent pages mapped as 4KB pages (each constituent page that is mapped must have already been prepared). At a minimum, on a page fault, the 4KB page that triggered the fault must be prepared immediately in order to allow the application to resume. However, upon a page fault, the OS can choose to prepare the entire physical superpage, only prepare the required 4KB page, or prepare the required 4KB page, allow the application to resume, and prepare the remaining 4KB pages later (either asynchronously or when they are accessed).

The three types of preparation — zeroing, copying, and file reading — have different costs, and so may impact the choice of when and how much of a physical superpage to prepare. Incremental preparation decreases page fault latency and minimizes unnecessary preparation for 4KB pages that may ultimately never get accessed. However, as the page is incrementally prepared, the constituent pages will be using 4KB mappings. In contrast, all at once preparation eliminates future page faults to the virtual memory region and allows for the immediate creation of a superpage mapping.

2.3 Superpage Mapping Creation

Once a physical superpage has been fully prepared, it must then be mapped as such in order to achieve address transla-

[1] https://github.com/rice-systems/quicksilver



*Allocation may fail because of memory fragmentation

*An incrementally prepared SP can be mapped as 4KB pages

*TLB coverage increases when caching created SP mappings

*4KB mappings can be created for some or all constituent 4KB pages

A: expect a virtual SP in the future
B: use individually for other purposes

Figure 1: The five events in the life cycle of a superpage (SP).

tion benefits. Before the superpage is mapped, the physical memory can still be accessed via 4KB mappings; afterwards, the OS loses the ability to track accesses and modifications at a 4KB granularity. Therefore, an OS may delay the creation of a superpage mapping if only some of the constituent pages are dirty in order to avoid unnecessary future I/O.

A superpage mapping is typically created upon a page fault, on either the initial fault to the memory region or a subsequent fault after the entire superpage has been prepared. However, if the physical superpage preparation is asynchronous, then its superpage mapping may also be created asynchronously. Note that on some architectures, *e.g.*, ARM, any 4KB mappings that were previously created must first be destroyed.

2.4 Superpage Mapping Destruction

Superpage mappings can be destroyed at any time, but must be destroyed whenever any part of the virtual superpage is freed or has its protection changed. After the superpage mapping is destroyed, 4KB mappings must be recreated for any constituent pages that have not been freed.

With superpage mappings, the OS cannot track whether constituent pages are accessed or modified. Therefore, in some scenarios, the OS may choose to preemptively destroy a superpage mapping and substitute 512 4KB mappings for it to enable finer-grained memory management. For example, when a clean superpage is first modified, the OS could choose to destroy the superpage mapping in order to only mark the single modified 4KB page as dirty, potentially reducing future I/O operations. This would require the OS to make a read-only superpage mapping and use the page fault caused by the write access to destroy the mapping and replace it with 4KB mappings. Similarly, the OS could choose to destroy a superpage mapping when under memory pressure to enable swapping pages at a finer granularity.

2.5 Physical Superpage Deallocation

Generally, a physical superpage is deallocated when an application frees some or all of the virtual superpage, when an application terminates, or when the OS needs to reclaim memory. If a superpage mapping exists, it must be destroyed before the physical superpage can be deallocated. Then, either the entire 2MB can be returned to the physical memory allocator or the physical superpage can be "broken" into 4KB pages. If the physical superpage is broken into its constituent 4KB pages, the OS can return a subset of those pages to the physical memory allocator. However, returning only a subset of the constituent pages increases memory fragmentation, decreasing the likelihood of future physical superpage allocations.

Before part or all of a physical superpage is returned to the physical memory allocator, any constituent pages that have been prepared but not freed must be preserved. Preservation typically happens in one of three ways. In-use pages can be kept rather than returned to the allocator, and 4KB mappings can be created to those pages. Alternatively, the in-use pages can be copied to other physical pages, allowing the entire physical superpage to be returned. The last option is to write the in-use pages to secondary storage before returning them.

3 State-of-the-art Designs

This section compares the state-of-the-art designs for transparent superpage management in FreeBSD, Linux, and recent research prototypes (Ingens [20] and HawkEye [24]), with a particular focus on how they manage the events described in the previous section.

3.1 FreeBSD

FreeBSD supports transparent superpages for all kinds of memory, including memory-mapped files and executables. It decouples physical superpage allocation from preparation by using a reservation-based memory allocator [23,29]. FreeBSD tries to allocate ("reserves") a physical superpage upon the first page fault to any aligned 2MB region. If physical superpages are available, they are allocated for any memory-mapped file exceeding 2MB in size. Anonymous memory always uses superpages if available, regardless of size, as anonymous memory is expected to grow.

Once a physical superpage is allocated for anonymous memory, only the 4KB page that caused the page fault is prepared, and a reservation entry is created to track all of the constituent pages. Any subsequent page fault to that 2MB region skips page allocation and simply prepares one additional 4KB page of the physical superpage. The physical superpage preparation finishes once all of its constituents have been prepared. For file-backed memory, the process is the same, except memory is prepared in 64KB batches to minimize I/O overhead.

FreeBSD creates superpage mappings synchronously during page faults. FreeBSD only creates a superpage mapping if the characteristics (*e.g.*, protection and modified state) of all the constituent 4KB mappings are the same. Identical protections are required for correctness; identical dirty states ensure that FreeBSD will not do unnecessary I/O to preserve the contents of the page when it is later deallocated.

Superpage mappings are destroyed on partial memory protection changes and partial unmappings. FreeBSD also preemptively destroys clean superpage mappings before modification. As a result, only one 4KB mapping is marked as dirty, instead of the entire superpage. Once the last clean 4KB page is modified, a dirty superpage mapping gets created.

FreeBSD defers physical superpage deallocation as long as possible in order to minimize memory fragmentation and preserve the availability of free physical superpages. However, under memory pressure, FreeBSD looks for a partially prepared physical superpage and breaks the corresponding reservation to allow the unused memory within that 2MB physical memory region to be reclaimed for other uses.

3.2 Linux

Linux's THP only uses superpages for anonymous memory and tries to allocate a physical superpage on the first page fault to a 2MB-aligned virtual memory region. If allocation fails and defragmentation is enabled (the default), it immediately does memory compaction via page migration to create a free physical superpage. This blocks the faulting process, increasing page fault latency. Under severe fragmentation, migration may still fail to create a free physical superpage.

Linux does all-at-once physical superpage preparation: the entire physical superpage is always zeroed right after being allocated. This increases the initial page fault latency, but may reduce the average latency [24]. After this preparation, a superpage mapping is immediately created. The superpage mapping will be destroyed if some or all of the superpage is unmapped or has its protection settings changed. Once some or all of the superpage has been freed, the physical superpage is deallocated and free memory is immediately reclaimed.

This "first-touch" superpage policy only allocates physical superpages at the time of the first page fault. However, Linux also includes a kernel daemon called "khugepaged", which asynchronously scans the system page tables. When it finds an aligned 2MB anonymous virtual memory region that contains at least one dirty 4KB mapping, khugepaged tries to allocate a physical superpage. If a free physical superpage exists, it acquires it; otherwise, it calls Linux's memory compaction to reclaim one by migrating pages.

Before preparing this physical superpage, khugepaged blocks accesses to the virtual 2MB region by blocking page faults within the region and destroying the existing 4KB mappings. It then prepares all of the physical superpage's constituent 4KB pages, one at a time. For a previously mapped page, the contents are copied. Previously unmapped pages are zeroed. Finally, it installs a superpage mapping.

Khugepaged's preparation is more costly than the first-touch preparation that occurs in a page fault. It blocks accesses to the 2MB region, causes TLB shootdowns, and pollutes CPU caches. As a result, it is allowed by default to allocate at most 8 superpages every 10 seconds (1.6 MB/s).

When an application partially frees memory within a superpage without unmapping the virtual memory (*e.g.*, `MADV_DONTNEED`), it triggers the destruction of the superpage mapping and the deallocation of the physical superpage. The remaining in-use memory then gets mapped as 4KB pages. However, when khugepaged scans this 2MB region, it will unnecessarily migrate the mapped memory into another allocated superpage and effectively reallocate the freed memory. It is precisely this behavior of khugepaged which has led to the severe memory bloating reported in recent work [20, 24].

3.3 Ingens and HawkEye

Recent state-of-the-art prototypes (Ingens [20] and HawkEye [24]) attempt to mitigate the page fault latency spikes incurred by Linux's first-touch superpage policy as well as the memory bloat incurred by khugepaged — behaviors which have led many to suggest that Linux's transparent superpage support be disabled for best performance.

Both systems disable Linux's first-touch policy, instead allocating, preparing, and mapping only a single 4KB page on a page fault. They then effectively modify khugepaged to more aggressively manage superpages.

Khugepaged's behavior differs in default Linux, Ingens, and HawkEye in terms of order, threshold, and rate for superpage creation. To prevent excessive memory bloat, Ingens increases the threshold to trigger creation of a superpage from one in-use 4KB page to 90% in-use, meaning there must be at least 460 4KB mappings in a 2MB region in order to create a superpage for that region.

Ingens maintains a list of candidate 2MB-aligned regions on page faults. As long as the list is not empty, Ingens keeps creating superpage mappings. However, asynchronous superpage creation introduces a fairness problem that the scanning order of page tables can lead to long delays for some processes. To alleviate this, Ingens prioritizes processes with fewer superpages. In addition, Ingens actively compacts non-referenced memory at an aggressive rate.

HawkEye uses the same threshold as default khugepaged: one 4KB page. Under memory pressure, it scans mapped superpages and makes their zero-filled 4KB pages copy-on-write to a single zero-filled page to reclaim memory.

HawkEye also maintains a list of candidate 2MB-aligned regions, but further weights them by their memory utilization, the process's resident size, sampled access frequency and

TLB overheads. HawkEye then creates a superpage mapping for the one with the most weight that is believed to bring the highest TLB overhead, called fine-grained superpage management in the paper [24]. It attempts to obtain considerable address translation benefits with fewer superpages.

HawkEye's fine-grained superpage management further consumes CPU resources besides the migration-based superpage mapping creations. To avoid interference with running processes, it uses the same promotion rate (1.6MB/s) as Linux's default khugepaged.

4 Analysis of Existing Designs

This section analyzes the designs for transparent superpage management described in the previous section and presents several novel observations about them. These observations motivate the design of Quicksilver.

Platforms. All designs were evaluated on an Intel E3-1245 v6-based server with maximum turbo performance and hyperthreading enabled. This server has 4 physical cores, 32GB DDR4 2400 ECC RAM, and a 256GB NVMe SSD. Linux version 4.3 was used, as both Ingens and HawkEye are based on that version. FreeBSD version 11.2 was used, upon which Quicksilver is built. Swapping is disabled under every OS.

Benchmarks. A large variety of benchmarks are evaluated. GUPS performs 2^{32} serial random memory accesses to 2^{30} 64-bit integers (8GB) [13]. Graphchi-PR, BlockSVM and ANN use out-of-core implementations to solve big-data tasks [21, 32]. Graphchi-PR computes 3 iterations of PageRank on the preprocessed Twitter-2010 dataset [19]. BlockSVM trains a classification model on the kdd2010-bridge dataset [28]. ANN randomly queries nearest neighbors on 2GB preprocessed hash tables. XSBench is a parallel computation kernel of the Monte Carlo neutron transport algorithm [30]. Canneal and freqmine are PARSEC benchmarks with large memory footprints [10]. Gcc, mcf, DSjeng and XZ are SPEC CPU2017 benchmarks with large memory footprints [11]. Buildkernel compiles the FreeBSD 11.2 kernel.

Graphchi-PR, XSBench, canneal and Buildkernel are multithreaded to fully utilize CPU resources. Cold and Warm are Redis workloads benchmarking throughput and tail latency from a separate client machine with 8 threads and 16 request pipelines. The Cold workload populates an empty Redis instance with 16GB of 4KB objects. The Warm workload queries the fully populated 16GB Redis instance with a set/get ratio of 5:5 using 4KB objects. Del-70, Del-50, Range-S and Range-XL are Redis workloads benchmarking memory consumption. Del-70 and Del-50 insert 2 million 8KB objects and randomly delete 70% and 50% of them, respectively. Range-S and Range-XL insert randomly sized objects from small and large size ranges, respectively. Detailed benchmark settings and scripts can be found in the Quicksilver repository.

Workload	Linux-4KB	Linux-noKhugepaged	Linux
Del-70	11.6 GB	11.7 GB	19.8 GB
Range-XL	14.4 GB	25.7 GB	30.7 GB

Table 1: Redis memory consumption. Linux-noKhugepaged disables khugepaged.

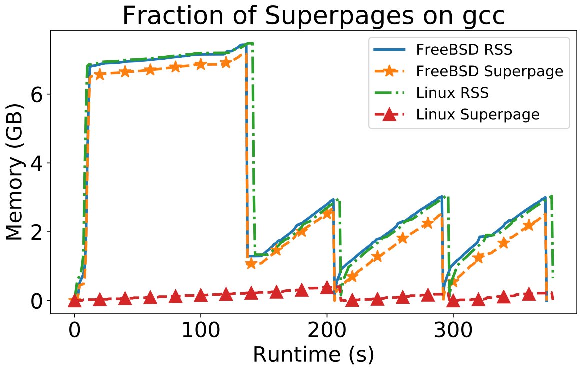


Figure 2: Linux's first touch policy fails to create superpages.

Observation 1: Coupling physical allocation, preparation, and mapping of superpages leads to memory bloat and fewer superpage mappings. It also is not compatible with transparent use of multiple superpage sizes.

Linux's first-touch policy couples physical superpage allocation, preparation and superpage mapping creation together. As a result, it enjoys two obvious benefits. First, it provides immediate address translation benefits, including shorter page walk time and increased TLB efficiency. Second, it eliminates a large number of page faults for a heavily utilized superpage. Therefore, it is usually the best mapping policy when there is abundant contiguous free memory.

However, this coupled policy has several drawbacks. First, it can easily bloat memory and waste time preparing underutilized superpages. In a microbenchmark that sparsely touches 30GB of anonymous memory, Linux's first-touch policy takes 1.4s to run and consumes 30GB compared to 0.06s and 0.2GB when disabling transparent huge pages. While such a corner case is rare when applications use `malloc` to dynamically allocate memory, it may still happen in a long-running server, *e.g.*, Redis. Table 1 shows that Linux's first touch policy bloats memory by 78% compared to Linux-4KB on the workload Range-XL, which inserts objects of random sizes ranging from 256B to 1MB.

Second, it misses chances to create superpage mappings when virtual memory grows. During a page fault, Linux cannot create a superpage mapping beyond the heap's end, so it installs a 4KB page which later prevents creation of a superpage mapping when the heap grows. Figure 2 shows such behavior for gcc [11], which includes three compilations. Linux's first-touch policy creates a few superpage mappings early in each compilation, but fails to create more as the heap grows. Instead, promotion-based policies can create more superpages,

Page Size	Anonymous	NVMe Disk	Spinning Disk
2MB	91 us	**1.7 ms**	**11 ms**
1GB	**46 ms**	**0.9 s**	**7.7 s**

Table 2: Page fault latency. Bold numbers are estimations.

e.g., FreeBSD and Linux's khugepaged.

Third, it cannot be extended to larger anonymous or file-backed superpages. Table 2 estimates the page fault latency on both 1GB anonymous superpages and 2MB/1GB file-backed superpages. Faulting a 2MB file-backed superpage on the NVMe disk costs 1.7ms and faulting a 1GB anonymous superpage takes 46ms. These numbers may cause latency spikes in server applications. Furthermore, it cannot easily determine which page size to use on first touch. This is arguably more of an immediate problem on ARM processors, which support both 64KB and 2MB superpages.

Observation 2: Asynchronous, out-of-place promotion alleviates latency spikes but delays physical superpage allocations.

Promotion-based policies can use 4KB mappings and later replace them with a superpage mapping. This allows for potentially better informed decisions about superpage mapping creation and can easily be extended to support multiple sizes of superpages. Specifically, there are two kinds of promotion policies, named out-of-place promotion and in-place promotion. They differ in whether previously prepared 4KB pages require migration when preparing a physical superpage.

Under out-of-place promotion a physical superpage is not allocated in advance, on a page fault a 4KB physical page is allocated that may neither be contiguous nor aligned with its neighbors. When the OS decides to create a superpage mapping, it must allocate a physical superpage, migrate mapped 4KB physical pages and zero the remaining ones. At this time, previously created 4KB mappings are no longer valid.

Linux and recent prototypes [20,24] perform asynchronous, out-of-place promotion to hide the cost of page migration. As discussed in Section 3.2, Linux includes khugepaged as a supplement to create superpage mappings for growing heaps. The steady, slow increase of Linux's superpages in Figure 2 is from khugepaged's out-of-place promotions. However, khugepaged can easily bloat memory. Table 1 shows a memory bloat from 11.6GB to 19.8GB on workload Del-70, which randomly deletes 70% of the objects. On workload Range-XL, it bloats memory from 25.7GB to 30.7GB.

Ingens and HawkEye [20, 24] disable Linux's first-touch policy and instead improve the behavior and functionality of khugepaged, motivated by avoiding latency spikes in server workloads. Under memory fragmentation, Linux tries to compact memory when it fails to allocate superpages, which blocks the ongoing page fault and leads to latency spikes. Ingens and HawkEye enhanced khugepaged and offloaded superpage allocations from the critical path, alleviating such latency spikes. So khugepaged works as their primary superpage management mechanism.

Workloads	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD
GUPS	0.87	0.84	0.28	0.88	0.96
Graphchi-PR	0.58	0.58	0.53	0.60	0.77
BlockSVM	0.81	0.79	0.73	0.81	0.96

Table 3: Speedup over Linux with unfragmented memory. All systems have worse performance than Linux.

However, out-of-place promotion delays physical superpage allocations and ultimately superpage mapping creation, because the OS must scan page tables to find candidate 2MB regions and schedule the background tasks to promote them. Table 3 compares in-place promotion (FreeBSD) with out-of-place promotion (Ingens and HawkEye) on applications where superpage creation speed is critical. While GUPS only involves random accesses, both Graphchi-PR and BlockSVM [21, 32] represent important real-life applications – using fast algorithms to process big data that cannot fit in memory. To better illustrate the problem, Ingens* and HawkEye* were tuned to be more aggressive, so that all 2MB regions containing at least one dirty 4KB mapping are candidates for promotion. Specifically, Ingens* uses a 0% utilization threshold instead of 90%; HawkEye* uses a 100% maximum CPU budget to promote superpages. However, Table 3 shows that FreeBSD consistently and significantly outperforms both of them. In other words, the most conservative in-place promotion policy creates superpage mappings faster than the most aggressive out-of-place promotion policy.

Observation 3: Reservation-based policies enable speculative physical page allocation, which enables the use of multiple page sizes, in-place promotion, and obviates the need for asynchronous, out-of-place promotion.

In-place promotion does not require page migration. It creates a physical superpage on the first touch, then incrementally prepares and maps its constituent 4KB pages without page allocation. Therefore, the allocation of a physical superpage is immediate, but its superpage mapping creation is delayed. To bypass 4KB page allocations, it requires a bookkeeping system to track allocated physical superpages, *e.g.*, FreeBSD's reservation system. On x86-64, after it substitutes a superpage mapping for the 4KB mappings, it need not flush the previous 4KB mappings from the TLB.

FreeBSD implements an in-place promotion policy based on its reservation system as described in Section 3.1. It conservatively creates superpage mappings to avoid making performance worse. Navarro, *et al.* reported negligible overheads from the reservation system [23].

FreeBSD immediately allocates physical superpages but delays superpage mapping creation, sacrificing some address translation benefits. Table 3 shows that Linux consistently outperforms FreeBSD when memory is unfragmented, though

	Linux-4KB	Linux
Frag-0	1.04 GB/s (5.6 ms)	1.34 GB/s (4.1 ms)
Frag-50	1.04 GB/s (5.7 ms)	0.92 GB/s (10.2 ms)

Table 4: Mean throughput and 95th latency of Redis Cold workload.

they created similar numbers of anonymous superpage mappings.

However, FreeBSD aggressively allocates physical superpages for anonymous memory. Upon a page fault of anonymous memory, it always speculatively allocates a physical superpage, expecting the heap to grow. This eliminates one of the primary needs for khugepaged in Linux. In Figure 2, FreeBSD has most of the memory quickly mapped as superpages, because most speculatively allocated physical superpages end up as fully-prepared pages.

Observation 4: Reservations and delaying partial deallocation of physical superpages fight fragmentation.

Superpages are easily fragmented on a long-running server. A few 4KB pages can consume a physical superpage, which benefits little if mapped as a superpage. Existing systems deal with memory fragmentation in three ways.

Linux compacts memory immediately when it fails to allocate a superpage. It tries to greedily use superpages, but risks blocking a page fault. Table 4 evaluated the performance of Redis. Under fragmentation, Linux obtains slightly higher throughput but much higher tail latency than Linux-4KB.

FreeBSD delays the partial deallocation of a physical superpage to increase the likelihood of reclaiming a free physical superpage. When individual 4KB pages get freed sooner, they land in a lower-ordered buddy queue and are more likely to be quickly reallocated for other purposes. Therefore, performing partial deallocations only when necessary due to memory pressure decreases fragmentation.

Ingens actively defragments memory in the background to avoid blocking page faults. It preferably migrates non-referenced memory, so that it minimizes the interference with running applications. As a result, Ingens generates fewer latency spikes compared with Linux [20]. These migrations, however, do consume processor and memory resources.

Observation 5: Bulk zeroing is more efficient on modern processors than repeated 4KB zeroing.

Modern OSes have abandoned asynchronous page zeroing because it usually degrades performance in a multiprocess situation. Furthermore, the introduction of ERMS (Enhanced REP MOVSB/STOSB) has accelerated page zeroing. However, existing OSes fail to fully exploit the benefits of ERMS support, because they still zero pages 4KB at a time. Modern CPUs can zero a 2MB page much faster with bulk zeroing, which calls the assembly language page zeroing code at a size larger than 4KB. Table 5 compares 2MB zeroing speed on five modern machines. Existing OSes take 84–409us to zero a 2MB superpage. After using a larger bulk size, the range is improved to 67–334us. Furthermore, these machines have a consistently short non-temporal (`movnti` or `clzero`) bulk zeroing latency (53–106us). The AMD Ryzen 7 2700X CPU achieves 53us with the highest CPU and DRAM frequency and its specific `clzero` implementation.

CPU (GHz)	DRAM	temporal			non-temporal		
Bulk Size:	(MHz)	4KB	32KB	2MB	4KB	32KB	2MB
E3-1231v3 (3.40)	1600	92	88	87	114	99	97
E3-1245v6 (3.70)	2400	84	67	65	92	74	71
E5-2640v3 (2.60)	1866	355	287	280	154	112	106
E5-2640v4 (2.40)	2133	409	334	325	163	113	106
R7-2700X (4.30)	2666	185	183	159	99	60	53

Table 5: 2MB page zeroing time (us) drops consistently using a larger bulk size.

5 Design and Implementation

This section describes Quicksilver, an improved transparent superpage management system based upon the observations from the previous section. To benefit from in-place promotions, Quicksilver is built upon FreeBSD's reservation-based superpage management strategy.

5.1 Design

Aggressive Physical Superpage Allocation. Section 4 shows that aggressive allocation on first touch (as done by Linux and FreeBSD) is effective. Moreover, Observation 3 shows that FreeBSD's reservation system allows speculative physical allocation for anonymous memory and creates even more superpages than Linux, as shown in Figure 2. Since it also supports multiple superpage sizes and avoids memory bloating, Quicksilver retains FreeBSD's reservation system: allocating physical superpages when virtual memory regions that may use superpages are first accessed. Allocation is performed synchronously to avoid page migrations.

The drawbacks of FreeBSD's use of reservations are twofold. First, FreeBSD delays preparation and mapping of superpages, resulting in lower performance than Linux in some scenarios, as shown in Table 3. However, this is not inherent in the use of reservations for allocation, but rather should be addressed via preparation and mapping policies. Second, holding underutilized physical superpages in reservations can prevent future superpage allocations. However, this is better resolved via deallocation policies that recognize and recover from such situations.

Hybrid Physical Superpage Preparation. Quicksilver strikes a balance between incremental and all-at-once preparation. Reservations are initially prepared incrementally. This minimizes the initial page fault latency, but loses prompt address translation benefits. Therefore, Quicksilver has an additional threshold, t. Once t 4KB pages get prepared, it prepares the remainder of the superpage all-at-once.

This reduces bloat, as discussed in Observation 1, because it does not immediately prepare and map the superpage. However, it enables address translation benefits sooner than waiting for the entire superpage to be accessed. The use of a threshold is further based on previous work showing that the utilization of physical superpages is largely bimodal [34]. Once more than about 64 4KB pages have been accessed, it is very likely that the physical superpage will eventually be fully populated (or very nearly so). Therefore, at that point, it is very likely to be beneficial to prepare the remainder of the page and create a superpage mapping for it. Motivated by Observation 5, bulk zeroing is used to accelerate page zeroing when zero-filling the remainder of the superpage.

Relaxed Superpage Mapping Creation. Once an entire physical superpage has been prepared, there is little downside to immediately creating a superpage mapping for anonymous memory, which is rarely, if ever, swapped in modern systems. Therefore, Quicksilver relaxes FreeBSD's current design — which does not create a superpage mapping if the accessed or modified states of the constituent pages differ — to always create a mapping once the physical superpage has been fully prepared, as do Linux, Ingens, and HawkEye.

For file-backed superpages, Quicksilver retains FreeBSD's write-protection mechanism to avoid extra disk I/O, but no longer examines if all constituent pages are accessed. Because memory-mapped files are usually prefetched 64KB at a time, file-backed superpages may not be fully accessed when they get fully prepared. By allowing different access bits, more file-backed superpage mappings can be created. Note that Linux and its variants do not use superpages at all for files.

On-demand Superpage Mapping Destruction. There is no reason to destroy a superpage mapping unless some or all of the memory within the superpage is freed, its protection is changed, or the physical superpage must be deallocated to reclaim memory. Therefore, Quicksilver maintains FreeBSD's policy of only destroying mappings in the aforementioned situations.

Preemptive Physical Superpage Deallocation. As discussed in Observation 4, delaying partial deallocation of physical superpages effectively limits fragmentation. However, to maximize the effectiveness of synchronous physical superpage allocation, there must be available superpages to allocate. Superpage availability can have a considerable impact on performance as was shown in Table 4. Therefore, Quicksilver maintains a target number of free physical superpages.

Underutilized reservations that are inactive for a long period are preemptively deallocated. These partially prepared physical superpages are not yet mapped as superpages, so the deallocation reduces memory bloat and recovers memory contiguity. Such preemptive deallocation copes well with hybrid preparation under a population threshold t. As a result, preemptive deallocation usually evacuates underutilized and less frequently accessed superpages.

This approach has three advantages. First, fewer pages are migrated. Second, the preemptive migration happens in the background, so it does not happen on the critical path of any OS function executed by the application. Finally, it is likely to have minimal impact on running processes, as it is operating on pages that come from less frequently accessed superpages.

5.2 Implementation

Quicksilver was implemented within FreeBSD 11.2. Quicksilver focuses on anonymous memory, with FreeBSD's superpage support for file-backed memory slightly improved (access bit equivalence is no longer required for promotion). This section further describes the page zeroing mechanism and the migration/deallocation daemon.

Hybrid Preparation. A physical superpage is incrementally prepared until it reaches a population threshold, t. Then the remainder of the physical superpage is prepared by zero-filling it. The system can do this either synchronously or asynchronously, named Sync-t and Async-t. Specifically, Async-t periodically scans the linked list of partially populated physical reservations and starts zero-filling from the most active ones reaching the population threshold t. Therefore, it incurs no fairness issue because the order is determined by physical allocation activity, not process IDs.

In both cases, zero-filling uses non-temporal stores. When using Sync-t, pages are zeroed using the largest bulk size possible, as motivated by Observation 5. Since zeroing is done by the page fault handler, the page fault handler can create a superpage mapping immediately after zeroing is complete. When using Async-t, 4KB pages are zeroed individually. While this yields lower zeroing performance, it reduces lock contention when operating on pages. Since zeroing is done asynchronously and independently of any process's virtual address space, a superpage mapping is not created until the first soft page fault after all pages have been zeroed.

Relaxed Mapping Creation. For anonymous memory, the superpage mapping creation condition is relaxed to ignore checking for dirty and access bits. This allows a superpage mapping to be created immediately after Sync-t completes the zero-filling (these pages are clean). For file-backed memory, superpage mappings are created on a soft page fault of a file-backed physical superpage. Default FreeBSD skips mapping creation because the access bits are inferred not to all be set when prefaulting the prefetched disk data. After relaxing the access bit checking, Quicksilver tries to create a superpage mapping at that point.

Preemptive Deallocation. Physical superpages are often underutilized [20, 34]. Given Observation 4, the system delays

Threads	Linux		FreeBSD	
	default	aggressive	default	emulate Linux ELF
1	1.05	1.19	1.15	1.16
8	**1.07**	0.91	1.11	**1.18**

Table 6: Canneal performance speedup. Only bold numbers are comparable.

Linux		FreeBSD		
default	aggressive	default	patched [1]	match Linux
1.01	1.24	1.02	1.07	**1.19**
0.4 K	8.2 K	0.0 K	1.2 K	**8.2 K**

Table 7: Throughput speedup and number of created superpage mappings of a Redis server populated by Del-70. Only bold numbers are comparable.

partial deallocation of physical superpages. However, to ensure that there are sufficient free physical superpages for future allocations, Quicksilver uses an evacuation daemon to reclaim free physical superpages by preemptively deallocating underutilized physical superpages.

The daemon maintains a target number of free physical superpages. It periodically scans the list of partially populated reservations and examines their inactive time, during which they are neither populated nor deallocated. If they are inactive for a long time, *e.g.*, 5 seconds, the daemon then reclaims a free physical superpage by migrating out its constituent 4KB pages. To avoid contention with running applications, the daemon is restricted to use a maximum memory bandwidth of 1GB/s. This is less than 5% of the evaluated machine.

6 Methodology

Fragmentation. Three fragmentation levels are modeled to mimic long-running servers, named Frag-0, Frag-50 and Frag-100. They represent situations from non-fragmented to severely-fragmented. Specifically, Frag-50 leaves 50% of the application's maximum resident memory as free superpages.

The three fragmentation levels are crafted by a user-space tool which works under a first-touch physical superpage allocation policy (available in both Linux and FreeBSD). It first fragments superpages until there is memory pressure, then starts over and fragments a target number of superpages. Unlike a previous memory fragmentation method [24] that only performs the latter step, Linux's memory compaction usually fails either in page faults or khugepaged. To fragment a superpage, the tool touches part of a 2MB-aligned virtual region and unmaps the untouched part. This will trigger a physical superpage allocation and force a partial deallocation, fragmenting that physical superpage.

Library Differences. FreeBSD dynamically links executable files with its natively shipped libc, while Linux uses GNU libc. This makes any performance comparison between FreeBSD and Linux unfair, because a different implementation of a standard function may change performance significantly. For example, the libc string library in FreeBSD 11.2 does not use ERMS optimizations, so memory copy-intense applications have worse performance. To remove this difference, applications were compiled and statically linked on Linux and then run on FreeBSD using FreeBSD's Linux system call emulation. Table 6 shows that natively compiled canneal on FreeBSD runs slower than emulated canneal, because of slower memory copying in dynamic array resizing. Although a libc library with ERMS optimizations could be ported from FreeBSD 12.0, this methodology ensures that the exact same binaries are run on all systems, eliminating any possible library differences.

There are three exceptions. GraphChi-PR uses `dlopen` to dynamically link the openmp library, so it cannot be statically compiled. Redis calls gettimeofday() very frequently, causing huge emulation overhead. Therefore, they are compiled natively on FreeBSD-based systems after porting the libc library from FreeBSD 12.0. They therefore may have minor library-induced performance differences between the Linux-based and FreeBSD-based systems. Lastly, FreeBSD's Linux emulation caused significant performance degradation on GUPS, because of cache misses resulting from an unaligned dynamically allocated data structure. To fix this, GUPS was modified to use `malloc_aligned`.

System Tuning. When there are idle CPUs, tuning Linux's khugepaged to be more aggressive can obtain better performance. Table 6 shows this in a single-threaded case. This tuning also yields higher throughput for single-threaded Redis, as shown in Table 7. However, performance degrades when the application uses all CPUs and competes with khugepaged, so Linux remains unchanged for the remainder of the evaluation.

FreeBSD 11.2 has suboptimal Redis performance due to three reasons. First, it uses a network socket buffer size suitable for 1Gbps NICs. Second, its libc has no ERMS optimizations, while memory copying dominates Redis's performance. Third, it is unlikely to repromote superpages after `MADV_FREE` (Redis uses `MADV_FREE` on FreeBSD to save page faults). Therefore, FreeBSD was tuned to use the correct buffer size for a 40Gbps NIC, and the libc library was ported from FreeBSD 12.0. Additionally, a recent patch [1] to FreeBSD was applied to increase the likelihood of superpage repromotion, creating 1.2K more superpage mappings in Table 7. The dirty bit requirement for anonymous memory was relaxed to match Linux's performance, creating 8.2K superpage mappings.

Ingens and HawkEye are evaluated with their default settings. Ingens promotes superpages with a 90%-utilization threshold. HawkEye promotes superpages at the speed of 1.6MB/s guided by performance counters. Ingens* and HawkEye* are aggressively tuned variants. Specifically, Ingens* uses a utilization threshold of 0% instead of 90% and enables 1GB/s proactive memory compaction. HawkEye* uses a 100% CPU maximum with a promotion threshold of 1.

7 Evaluation

Four variants of Quicksilver are considered, named Sync-1, Sync-64, Async-64 and Async-256. They all handle the five superpage events similarly except for superpage preparation. Therefore, for clarity they are named after their preparation policies. These four variants represent reasonable design points in the Sync-*t* and Async-*t* space, and use the same 1GB/s active defragmentation daemon. They share the same library and system tunings with FreeBSD. All performance numbers are the mean of three runs.

7.1 Non-fragmented (Frag-0) Performance

Sync-1 vs. Linux. Sync-1 uses the same superpage preparation and mapping policy for anonymous memory as Linux. With no fragmentation, Tables 8 and 9 show that they perform similarly. However, there are two notable differences. First, Sync-1 speculatively allocates superpages for growing heaps, which allows it to outperform Linux on canneal and gcc. Their similar speedups on reservation-based systems validate Observation 3. Second, Sync-1 creates file-backed superpages and outperforms Linux on ANN and Graphchi-PR.

Promotion Speed. Under Frag-0, FreeBSD often outperforms Ingens, HawkEye and their aggressively tuned variants, as shown in Table 8. This validates Observation 2, as the issue is that out-of-place promotion has a slower promotion speed. Furthermore, as shown in Table 9, on the Redis Cold workload, Ingens, HawkEye and their aggressively tuned variants even show a slight degradation compared to Linux-4KB. These systems introduce some noticeable interference with running applications when they manage superpages in the background.

Sync-64 mostly outperforms Async-64, because Async-64 zeros pages in the background which can cause interference. The comparable performance of Sync-64 and Sync-1 shows that less aggressive preparation and mapping policies can achieve comparable results to immediately mapping superpages on first touch.

7.2 Performance Under Fragmentation

Table 9 shows that Linux obtains a much higher tail latency on the Redis Cold workload under Frag-50/100 than Linux-4KB, because its on-allocation defragmentation significantly increases page fault latency. In contrast, FreeBSD does not actively defragment memory, so it generates no latency spikes.

Ingens and HawkEye offload superpage allocation from page faults and compact memory in the background, so they reduce interference and generate few latency spikes on the Redis Cold workload. Furthermore, as shown in Table 8, their speedup over Linux increases as fragmentation increases. However, HawkEye does not achieve the same speedups on XSBench that were reported in the original paper [24], because in these application runs, most of its memory compaction fails and its important data was not allocated at the high end of the address space.

The four variants of Quicksilver all consistently perform well on both non-server and server workloads, because their background defragmentation not only avoids increasing page fault latency, but also succeeds in recovering unfragmented performance. Specifically, on the Redis Cold workload, Sync-1 maintained the highest throughput (1.31 GB/s) while providing low (4.5 ms) tail latency under Frag-100. However, the per-second background scanning of the evacuation daemon may fail to improve performance when applications quickly touch all of their memory in the beginning (*e.g.* GUPS and ANN). As a result, there is high performance variation on GUPS and ANN performance is not improved over other systems, as shown in Table 8.

Graphchi-PR. On all applications in Table 8, the Sync-*t* and Async-*t* systems all match or outperform Linux. Since Graphchi-PR is an important and real-world task, it is selected to elaborate how the design choices described in Section 5 contribute to the 2.18 speedup of Sync-1 under Frag-100.

Under Frag-100, Async-64 obtains a speedup of 1.68, which is higher than the 1.15 speedup obtained by Ingens* on Graphchi-PR. When Graphchi-PR terminated, Ingens* has a total of 1,926 (mean of 3 runs) free physical superpages, but Async-64 has 11,955 free physical superpages. Although they have the same memory bandwidth budget (1GB/s) for active defragmentation, Quicksilver's evacuation daemon defragments memory more efficiently by identifying inactive fragmented superpages. The in-place promotions further contribute to the higher speedup of Async-64. When memory is not fragmented, Async-64 obtains a speedup of 0.83, higher than all other non-Quicksilver systems.

Sync-64 obtains an even higher speedup of 2.11. The shared evacuation daemon allows both Async-64 and Sync-64 to allocate a similar number of superpages, but the synchronous all-at-once preparation implemented by bulk zeroing in Sync-64 efficiently removes the delay of creating superpages. With the same number of superpages, Sync-64 is able to reduce page walk pending cycles by 76%. The highest speedup is obtained by Sync-1 with a more aggressive promotion threshold.

7.3 Memory Bloat

All systems suffer less than 1% memory bloat compared to Linux-4KB on the applications shown in Table 8. However, long-running servers may still suffer from memory bloat. When applications frequently allocate and deallocate memory, an aggressive superpage preparation policy may preemptively prepare a superpage and sacrifice free memory for minor address translation benefits, ultimately creating false memory pressure.

Frag-0	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	0.87	0.58	0.81	0.98	1.00	0.95	0.99	1.00	0.99	0.99	0.96
Ingens*	0.84	0.58	0.79	0.97	0.97	0.92	0.99	1.01	0.96	0.99	0.92
HawkEye	0.28	0.53	0.73	0.88	1.00	0.95	0.99	0.99	0.94	0.86	0.90
HawkEye*	0.88	0.60	0.81	0.98	1.00	0.97	1.00	0.99	0.97	0.99	0.94
FreeBSD	0.96	0.77	0.96	0.99	0.98	1.14	1.00	1.05	0.99	1.00	0.99
Sync-1	0.99	1.07	1.00	1.00	1.07	1.14	0.99	1.05	1.00	1.00	1.00
Sync-64	0.98	1.05	1.00	1.00	1.08	1.14	0.99	1.05	1.00	1.00	1.00
Async-64	0.96	0.83	0.97	0.99	1.08	1.14	1.00	1.05	1.00	1.00	0.99
Async-256	0.96	0.82	0.97	0.99	1.08	1.14	0.99	1.05	0.99	1.00	0.99

Frag-50	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	0.98	0.71	0.82	1.01	1.00	0.99	1.00	1.00	1.00	1.00	0.99
Ingens*	1.24	0.73	0.86	1.00	0.98	1.00	0.99	1.02	0.99	1.04	0.97
HawkEye	0.62	0.68	0.77	0.91	1.00	0.96	1.00	0.99	0.97	0.92	0.94
HawkEye*	0.89	0.68	0.80	1.00	0.99	0.99	1.00	0.99	0.99	0.98	0.99
FreeBSD	0.98	0.94	0.89	1.02	0.97	1.01	1.00	1.05	1.01	1.02	1.01
Sync-1	2.04(0.08)	1.37	1.04	1.03	1.04	1.17	1.00	1.05	1.03	1.05	1.05
Sync-64	2.01	1.32	1.06	1.03	1.04	1.18	1.00	1.05	1.03	1.06	1.05
Async-64	2.11	1.06	1.02	1.03	1.03	1.17	1.00	1.05	1.03	1.06	1.04
Async-256	2.11	1.05	1.02	1.03	1.03	1.17	1.00	1.05	1.03	1.06	1.04

Frag-100	GUPS	Graphchi-PR	BlockSVM	XSBench	ANN	canneal	freqmine	gcc	mcf	DSjeng	XZ
Ingens	1.02	1.13	0.86	1.04	1.00	1.00	1.00	1.00	1.01	1.01	1.02
Ingens*	1.30	1.15	0.88	1.13	0.99	1.06	1.00	1.02	1.03	1.08	1.06
HawkEye	0.97	1.11	0.85	1.03	1.00	1.01	1.00	1.00	0.99	0.97	1.02
HawkEye*	0.96	1.11	0.84	1.03	1.00	1.01	1.00	0.99	0.99	0.97	1.01
FreeBSD	0.96	1.10	0.85	1.04	0.98	1.05	1.00	1.00	1.00	1.04	1.02
Sync-1	2.35(0.30)	2.18	1.12	1.07	1.04	1.12	1.00	1.05	1.02	1.10	1.14
Sync-64	2.29(0.14)	2.11	1.13	1.07	1.01	1.12	1.00	1.05	1.05	1.11	1.14
Async-64	1.91(0.21)	1.68	1.11	1.06	0.98	1.12	1.00	1.05	1.05	1.11	1.13
Async-256	2.10(0.22)	1.65	1.10	1.08	0.98	1.16	1.00	1.06	1.05	1.08	1.13

Table 8: Performance speedup over Linux under three fragmentation levels. Red boxes indicate that the system performs worse than Linux on that application. The normalized standard deviation of runtime is no greater than 5% unless specified in parentheses.

Table 10 compares the memory consumption of four Redis workloads. Among these workloads, Linux bloats memory the most, consistent with previous findings [20]. However, Sync-1 exhibits lower memory consumption than Linux despite similar policies. In fact, it is khugepaged that bloats memory. When a partially deallocated superpage is scanned, it allocates the memory back to recreate a superpage, undermining the application's efforts to free and defragment memory.

All systems other than Linux limit memory consumption for the first three workloads; they only really differ on Range-XL. HawkEye, FreeBSD, and Async-256 exhibit the lowest memory consumption on Range-XL, whereas the other systems bloat memory by 40–60%. HawkEye stops allocating superpages when the TLB overhead is minor, FreeBSD only promotes fully utilized superpages, and Async-256 has a conservative promotion threshold.

Sync-1 vs. Sync-64 Besides bloating memory, aggressive preparation policies may cause excessive creation of superpages. This is common when many small processes get forked. For example, Table 11 shows what happens in a 9-threaded compilation of the FreeBSD kernel. Sync-1 creates more than 200k superpages, while the less aggressive Sync-64 only creates around 100k. Over half of the superpages created by Sync-1 had less than 13% utilization. Consequently, Sync-1 spends 13.9% more system time preparing them, which outweighs their benefits. In a long running server, using an aggressive policy like Sync-1 could waste both power and memory contiguity by creating underutilized superpages. In contrast, Sync-64 avoids such cases and suffers from less performance degradation than Sync-1 in both Table 8 and Table 9. Therefore, it is more preferable for long-running servers.

8 Related Work

Direct segments have been proposed as a supplement to existing page-based address translation for large-memory applications [9, 14, 18]. While they are effective at reducing the cost of address translation, they are limited to systems that allocate nearly all of the system memory to a single application with the same access rights. While these ideas can be generalized to some degree, they ultimately limit the flexibility of the OS to allocate and use physical memory.

Automatic TLB entry coalescing to increase the effective

Cold	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Frag-0	1.04(5.6)	1.34(4.1)	1.00(5.9)	0.98(6.3)	1.00(5.9)	1.00(5.8)	1.11(6.1)	1.26(4.5)	1.20(4.8)	1.10(6.0)	1.11(6.0)
Frag-50	1.04(5.7)	0.92(10.2)	0.95(5.9)	0.97(5.9)	1.02(5.9)	1.03(5.8)	1.04(6.2)	1.25(4.5)	1.27(4.7)	1.09(6.0)	1.09(6.0)
Frag-100	1.07(5.6)	0.81(9.9)	0.94(6.1)	0.97(6.1)	1.00(5.8)	1.02(5.8)	0.98(6.5)	1.31(4.5)	1.26(4.6)	1.14(5.9)	1.08(5.9)

Warm	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Frag-0	1.06(6.5)	1.32(5.2)	1.23(5.7)	1.21(5.8)	1.03(6.7)	1.06(6.5)	1.30(5.6)	1.32(5.5)	1.31(5.5)	1.31(5.5)	1.30(5.6)
Frag-50	1.07(6.5)	1.17(5.9)	1.09(6.4)	1.19(5.8)	1.03(6.7)	1.05(6.7)	1.18(6.1)	1.32(5.5)	1.32(5.5)	1.31(5.5)	1.31(5.5)
Frag-100	1.07(6.5)	1.16(5.9)	1.01(6.9)	1.09(6.4)	1.05(6.6)	1.07(6.5)	1.10(6.6)	1.33(5.4)	1.34(5.5)	1.33(5.4)	1.31(5.5)

Table 9: Redis throughput (GB/s) and 95th latency (ms) of workloads Cold and Warm. Numbers in parentheses are 95th latencies. The maximum standard deviation is 0.04GB/s for throughput and 0.57ms for 95th latency.

Workload	Linux-4KB	Linux	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD	Sync-1	Sync-64	Async-64	Async-256
Del-70	11.6	19.8	11.6	11.7	11.6	11.6	11.6	11.6	11.6	11.6	11.6
Del-50	16.7	19.8	16.8	16.8	16.7	16.9	16.7	16.8	16.8	16.8	16.8
Range-S	14.3	15.6	16.0	15.6	14.9	14.5	14.3	15.6	15.6	15.3	15.1
Range-XL	14.4	30.7	22.7	23.3	15.7	20.6	14.9	23.1	20.9	19.5	15.9

Table 10: Redis memory consumption (GB) of four workloads. Khugepaged further bloats memory in Linux.

Buildkernel	real	user	sys	# SP	# PF
Sync-1	197.7	1409.4	89.4	200.5 K	5.3 M
Sync-64	196.9	1408.8	78.5	99.6 K	10.3 M
FreeBSD	203.7	1436.7	98.0	36.9 K	30.2 M

Table 11: Runtime (seconds) and numbers of superpages and page faults of compiling the FreeBSD 11.2 kernel.

reach of the TLB has been proposed and implemented [26,27]. Essentially, a page walk will load multiple 4KB mappings found in the same cache line. If these mappings refer to contiguous pages and have identical access privileges, then they are merged into a single TLB entry. Although TLB entry coalescing occurs automatically in hardware, it nonetheless requires the OS to allocate physically contiguous memory. AMD Ryzen processors do such coalescing [12].

A large body of work has shown that using superpages can reduce the cost of address translation. Originally, OS support for superpages required the administrator to manually control the use of superpages. For example, Linux has long supported *persistent huge pages* [4]. A huge page pool with a static number of huge pages must be allocated by the administrator before running applications. The persistent huge pages are pinned in memory and can only be used via specific flags to `mmap` system calls. Superpage support in Windows and OS X are similar to Linux persistent huge pages [3,6].

To eliminate the need for manual control, FreeBSD, Linux, and many research prototypes have explored transparent superpage support, as described in Section 3. This support has been extensively described and studied [16,17,20,23,24,29]. As this transparent support for superpages has become widely available in production OSes, many people have argued that effectively handling all of the issues that can arise still requires further improvements to OS memory management support [15–17,20,22,24,25]. For example, some of these people have worked to improve Linux's superpage management by decreasing memory fragmentation and more carefully allocating physical superpages using Linux's idle page tracking mechanisms [20,22,24,25,31]. Others have shown that it is beneficial to decrease memory fragmentation and increase the contiguity of physical memory. To achieve this, several efforts have focused on minimizing migration and reducing its performance impact, while still attempting to reduce fragmentation and increase contiguity [7,8,22,25,31]. The deprecated lumpy reclaim from Linux was also developed to increase contiguity [2]. It reclaims a 2MB superpage by finding an inactive 4KB page and swaps out all dirty 4KB pages inside the 2MB block. Because these dirty 4KB pages may also contain active ones, swapping them out may hurt performance instead. Besides efforts on anonymous superpages, Zhou, *et al.* augmented FreeBSD to synchronously page-in code and pad code sections to create more code superpages [33].

9 Conclusions

This paper has performed a comprehensive analysis of superpage management mechanisms and policies. The explicit enumeration of the five events involved in the life of a superpage provides a framework around which to compare and contrast superpage management policies. This framework and analysis yielded five key observations about superpage management that motivated Quicksilver's innovative design. Quicksilver achieves the benefits of aggressive superpage allocation, while mitigating the memory bloat and fragmentation issues that arise from underutilized superpages. Both the Sync-1 and Sync-64 variants of Quicksilver are able to match or beat the performance of existing systems in both lightly and heavily fragmented scenarios, in terms of application performance, tail latency, and memory bloat. However, Sync-64 is preferable for long-running servers, as it does not aggressively create underutilized superpages.

References

[1] FreeBSD MADV_FREE heuristics. https://svnweb.freebsd.org/base?view=revision&revision=350463. Viewed 2020-05-31.

[2] Linux's lumpy reclaim. https://lkml.org/lkml/2012/3/28/323. Viewed 2020-05-31.

[3] OS X superpage support. https://www.unix.com/man-page/osx/2/mmap/. Viewed 2020-05-31.

[4] Persistent huge pages in Linux. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt. Viewed 2020-05-31.

[5] Transparent huge pages in Linux. https://www.kernel.org/doc/Documentation/vm/transhuge.txt. Viewed 2020-05-31.

[6] Windows large page support. https://docs.microsoft.com/en-us/windows/desktop/memory/large-page-support. Viewed 2020-05-31.

[7] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: Enabling application-transparent support for multiple page sizes in throughput processors. *ACM SIGOPS Operating Systems Review*, 51(1):27–44, 2018.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 237–248, 2013.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[11] James Bucek, Klaus-Dieter Lange, et al. SPEC CPU2017: next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42. ACM, 2018.

[12] Mike Clark. A new x86 core architecture for the next generation of computing. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–19. IEEE, 2016.

[13] II Earl Joseph. Gups (giga-updates per second) benchmark. *URL http://www. dgate. org/˜ brg/files/dis/gups*, 2000.

[14] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 178–189, 2014.

[15] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, 2014.

[16] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management*, pages 41–50. ACM, 2008.

[17] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *International Symposium on Computer Architecture*, pages 293–310. Springer, 2010.

[18] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 66–78, 2015.

[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. AcM, 2010.

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 705–721, 2016.

[21] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46, 2012.

[22] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 121–131. ACM, 2019.

[23] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[24] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained OS support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360. ACM, 2019.

[25] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *ACM SIGPLAN Notices*, volume 53, pages 679–692. ACM, 2018.

[26] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 558–567, 2014.

[27] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 258–269, 2012.

[28] J Stamper, A Niculescu-Mizil, S Ritter, GJ Gordon, and KR Koedinger. Bridge to algebra 2008–2009. *Challenge data set from KDD Cup*, 2010.

[29] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994.*, pages 171–182, 1994.

[30] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis.

[31] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 698–710, 2019.

[32] Hsiang-Fu Yu, Cho-Jui Hsieh, Kai-Wei Chang, and Chih-Jen Lin. Large linear classification when data cannot fit in memory. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[33] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–116. IEEE, 2019.

[34] Weixi Zhu. Exploring superpage promotion policies for efficient address translation. Master's thesis, Rice University, 6100 Main St, Houston, TX 77005, 2019.

Effectively Prefetching Remote Memory with Leap

Hasan Al Maruf
University of Michigan

Mosharaf Chowdhury
University of Michigan

Abstract

Memory disaggregation over RDMA can improve the performance of memory-constrained applications by replacing disk swapping with remote memory accesses. However, state-of-the-art memory disaggregation solutions still use data path components designed for slow disks. As a result, applications experience remote memory access latency significantly higher than that of the underlying low-latency network, which itself can be too high for many applications.

In this paper, we propose Leap, a prefetching solution for remote memory accesses due to memory disaggregation. At its core, Leap employs an online, majority-based prefetching algorithm, which increases the page cache hit rate. We complement it with a lightweight and efficient data path in the kernel that isolates each application's data path to the disaggregated memory and mitigates latency bottlenecks arising from legacy throughput-optimizing operations. Integration of Leap in the Linux kernel improves the median and tail remote page access latencies of memory-bound applications by up to 104.04× and 22.62×, respectively, over the default data path. This leads to up to 10.16× performance improvements for applications using disaggregated memory in comparison to the state-of-the-art solutions.

1 Introduction

Modern data-intensive applications [5, 29, 30, 70] experience significant performance loss when their complete working sets do not fit into the main memory. At the same time, despite significant and disproportionate memory underutilization in large clusters [62, 78], memory cannot be accessed beyond machine boundaries. Such unused, stranded memory can be leveraged by forming a cluster-wide logical memory pool via *memory disaggregation*, improving application-level performance and overall cluster resource utilization [11, 45, 48].

Two broad avenues have emerged in recent years to expose remote memory to memory-intensive applications. The first requires redesigning applications from the ground up using RDMA primitives [15, 22, 36, 49, 59, 63, 77]. Despite its efficiency, rewriting applications can be cumbersome and may not even be possible for many applications [10]. Alternatives rely on well-known abstractions to expose remote memory; e.g., distributed virtual file system (VFS) for remote file access [10] and distributed virtual memory management (VMM) for *remote memory paging* [28, 32, 45, 46, 65].

Because disaggregated remote memory is slower, keeping hot pages in the faster local memory ensures better performance. Colder pages are moved to the far/remote memory as needed [9, 32, 45]. Subsequent accesses to those cold pages go through a slow data path inside the kernel – for instance, our measurements show that an average 4KB remote page access takes close to 40 μs in state-of-the-art memory disaggregation systems like Infiniswap. Such high access latency significantly affects performance because memory-intensive applications can tolerate at most single μs latency [28, 45]. Note that the latency of existing systems is many times more than the 4.3 μs average latency of a 4KB RDMA operation, which itself can be too high for some applications.

In this paper, we take the following position: *an ideal solution should minimize remote memory accesses in its critical path as much as possible*. In this case, a *local page cache* can reduce the total number of remote memory accesses – a cache *hit* results in a sub-μs latency, comparable to that of a local page access. An effective prefetcher can proactively bring in correct pages into the cache and increase the cache hit rate.

Unfortunately, existing prefetching algorithms fall short for several reasons. First, they are designed to reduce disk access latency by prefetching sequential disk pages in large batches. Second, they cannot distinguish accesses from different applications. Finally, they cannot quickly adapt to temporal changes in page access patterns within the same process. As a result, being optimistic, they pollute the cache with unnecessary pages. At the same time, due to their rigid pattern detection technique, they often fail to prefetch the required pages into the cache before they are accessed.

In this paper, we propose Leap, an online prefetching solution that minimizes the total number of remote memory accesses in the critical path. Unlike existing prefetching algorithms that rely on strict pattern detection, Leap relies on approximation. Specifically, it builds on the Boyer-Moore majority vote algorithm [17] to efficiently identify remote memory access patterns for each individual process. Relying on an approximate mechanism instead of looking for trends in strictly consecutive accesses makes Leap resilient to short-term irregularities in access patterns (e.g., due to multi-threading). It also allows Leap to perform well by detecting trends only from remote page accesses instead of tracing the full virtual memory footprint of an application, which demands continuous scanning and logging of the hardware access bits of the whole virtual address space and results in high CPU and memory overhead. In addition to identifying

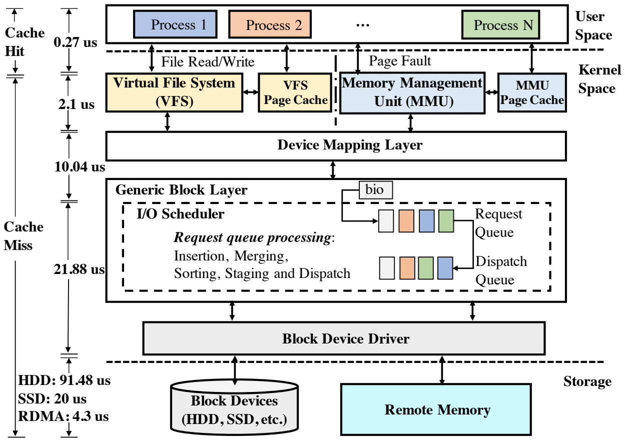


Figure 1: High-level life cycle of page requests in Linux data path along with the average time spent in each stage.

the majority access pattern, Leap determines how many pages to prefetch following that pattern to minimize cache pollution.

While reducing cache pollution and increasing the cache hit rate, Leap also ensures that the host machine faces minimal memory pressure due to the prefetched pages. To move pages from local to remote memory, the kernel needs to scan through the entire memory address-space to find eviction candidates – the more pages it has, the more time it takes to scan. This increases the memory allocation time for new pages. Therefore, alongside a background LRU-based asynchronous page eviction policy, Leap eagerly frees up a prefetched cache just after it gets hit and reduces page allocation wait time.

We complement our algorithm with an efficient data path design for remote memory accesses that is used in case of a cache *miss*. It isolates per-application remote traffic and cuts inessentials in the end-host software stack (e.g., the block layer) to reduce host-side latency and handle a cache miss with latency close to that of the underlying RDMA operations.

Overall, we make the following contributions in this paper:

- We analyze the data path latency overheads for disaggregated memory systems and find that existing data path components can not consistently support single μs 4KB page access latency (§2).
- We propose Leap, a novel online prefetching algorithm (§3) and an eager prefetch cache eviction policy along with a leaner data path, to improve remote I/O latency.
- We implement Leap on Linux Kernel 4.4.125 as a separate data path for remote memory access (§4). Applications can choose either Linux's default data path for traditional usage or Leap for going beyond the machine's boundary using unmodified Linux ABIs.
- We evaluate Leap's effectiveness for different memory disaggregation frameworks. Leap's faster data path and effective cache management improve the median and tail 4KB page access latency by up to 104.04× and 22.62× for micro-benchmarks (§5.1) and by 1.27–10.16× for real-world memory-intensive applications with production workloads (§5.3).
- We evaluate Leap's prefetcher against practical real-time prefetching techniques (Next-K Line, Stride, Linux Read-ahead) and show that simply replacing the default Linux prefetcher with Leap's prefetcher can provide application-level performance benefit (1.1–3.36× better) even when they are paging to slower storage (e.g., HDD, SSD) (§5.2).

2 Background and Motivation

2.1 Remote Memory

Memory disaggregation systems logically expose unused cluster memory as a global memory pool that is used as the slower memory for machines with extreme memory demand. This improves the performance of memory-intensive applications that have to frequently access slower memory in memory-constrained settings. At the same time, the overall cluster memory usage gets balanced across the machines, decreasing the need for memory over-provisioning per machine.

Access to remote memory over RDMA without significant application rewrites typically relies on two primary mechanisms: disaggregated VFS [10], that exposes remote memory as files and disaggregated VMM for remote memory paging [32, 45, 65]. In both cases, data is communicated in small chunks or pages. In case of remote memory as files, pages go through the file system before they are written to/read from the remote memory. For remote memory paging and distributed OS, page faults cause the virtual memory manager to write pages to and read them from the remote memory.

2.2 Remote Memory Data Path

State-of-the-art memory disaggregation frameworks depend on the existing kernel data path that is optimized for slow disks. Figure 1 depicts the major stages in the life cycle of a page request. Due to slow disk access times – average latencies for HDDs and SSDs range between 4–5 ms and 80–160 μs, respectively – frequent disk accesses have a severe impact on application throughput and latency. Although the recent rise of memory disaggregation is fueled by the hope that RDMA can consistently provide single μs 4KB page access latency [11, 28, 32], this is often a wishful thinking in practice [79]. Blocking on a page access – be it from HDD, SSD, or remote memory – is often unacceptable.

To avoid blocking on I/O, race conditions, and synchronization issues (e.g., accessing a page while the page out process is still in progress), the kernel uses a page cache. To access a page from slower memory, it is first looked up in the appropriate cache location; a *hit* results in almost memory-speed page access latency. However, when the page is not found in the cache (i.e., a *miss*), it is accessed through a costly block device I/O operation that includes several queuing and batching stages to optimize disk throughput by merging multiple contiguous smaller disk I/O requests into a single large re-

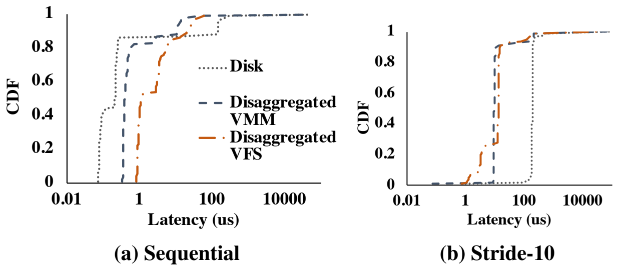


Figure 2: Data path latencies for two access patterns. Memory disaggregation systems have some constant implementation overheads that cap their minimum latency to around 1 μs.

quest. On average, these batching and queuing operations cost around 34 μs and over a few milliseconds at the tail. As a result, a cache miss leads to more than 100× slower latency than a hit; it also introduces high latency variations. For microsecond-latency RDMA environments, this unnecessary wait-time has a severe impact on application performance.

2.3 Prefetching in Linux

Linux tries to store files on the disk in adjacent sectors to increase sequential disk accesses. The same happens for paging. Naturally, existing prefetching mechanisms are designed assuming a sequential data layout. The default Linux prefetcher relies on the last two page faults: if they are for consecutive pages, it brings in several sequential pages into the page cache; otherwise, it assumes that there are no patterns and reduces or stops prefetching. This has several drawbacks. First, whenever it observes two consecutive paging requests for consecutive pages, it over-optimistically brings in pages that may not even be useful. As a result, it wastes I/O bandwidth and causes *cache pollution* by occupying valuable cache space. Second, simply assuming the absence of any pattern based on the last two requests is over-pessimistic. Furthermore, all the applications share the same swap space in Linux; hence, pages from two different processes can share consecutive places in the swap area. An application can also have multiple, inter-leaved stride patterns – for example, due to multiple concurrent threads. Overall, considering only the last two requests to prefetch a batch of pages falter on both respects.

To illustrate this, we measure the page access latency for two memory access patterns: (a) **Sequential** accesses memory pages sequentially, and (b) **Stride-10** accesses memory in strides of 10 pages. In both cases, we use a simple application with its working set size set to 2GB. For disaggregated VMM, it is provided 1GB memory to ensure that 50% of its access cause paging. For disaggregated VFS, it performs 1GB remote write and then another 1GB remote read operations.

Figure 2 shows the latency distributions for 4KB page accesses from disk and disaggregated remote memory for both of the access patterns. For a prefetch size of 8 pages, both perform well for the **Sequential** pattern; this is because 80% of the requests hit the cache. In contrast, we observe significantly higher latency in the **Stride-10** case because all the requests miss the page cache due to the lack of consecutiveness in successive page accesses. By analyzing the latency breakdown inside the data path for **Stride-10** (as shown in Figure 1), we make two key observations. First, although RDMA can provide significantly lower latency than disk (4.3μs vs. 91.5μs), RDMA-based solutions do not benefit as much from that (38.3μs vs. 125.5μs). This is because of the significant data path overhead (on average 34μs) to prepare and batch a request before dispatching it. Significant variations in the preparation and batching stages of the data path cause the average to stray far from the median. Second, the existing sequential data layout-based prefetching mechanism fails to serve the purpose in the presence of diverse remote page access patterns. Solutions based on fixed stride sizes also fall short because stride sizes can vary over time within the same application. Besides, there can be more complicated patterns beyond stride or no repetitions at all.

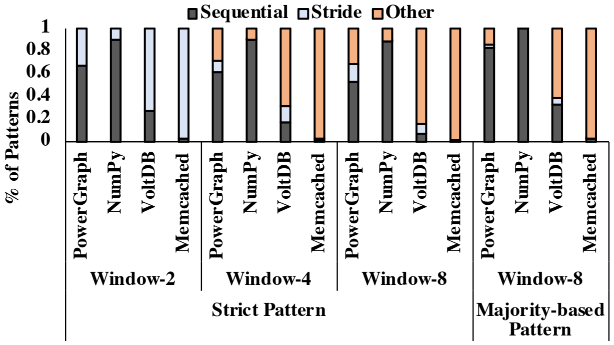


Figure 3: Fractions of sequential, stride, and other access patterns in page fault sequences of length X (Window-X).

Shortcoming of Strict Pattern Finding for Prefetching Figure 3 presents the remote page access patterns of four memory-intensive applications during page faults when they are run with 50% of their working sets in memory (more details in Section 5.3). Here, we consider all page fault sequences within a window of size $X \in \{2,4,8\}$ in these applications. Therefore, we divide the page fault scenarios into three categories: *sequential* when all pages within the window of X are sequential pages, *stride* when the pages within the window of X have the same stride from the first page, and *other* when it is neither sequential nor stride.

The default prefetcher in Linux finds strict sequential patterns in window size $X = 2$ and tunes up its aggressiveness accordingly. For example, page faults in PowerGraph and VoltDB follow 67% and 27% sequential pattern within window size $X = 2$, respectively. Consequently, for these two applications, Linux optimistically prefetches many pages into the cache. However, if we look at the $X = 8$ case, the percentage of sequential pages within consecutive page faults goes down to 53% and 7% for PowerGraph and VoltDB, respectively. Meaning, for these two applications, 14–20% of the prefetched pages are not consumed immediately. This creates

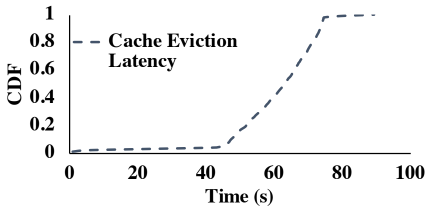


Figure 4: Due to Linux's lazy cache eviction policy, page caches waste the cache area for significant amount of time.

unnecessary memory pressure and might even lead to cache pollution. At the same time, all non-sequential patterns in the $X = 2$ case fall under the stride category. Considering the low cache hit rate, Linux pessimistically decreases/stops prefetching in those cases, which leads to a stale page cache.

Note that strictly expecting all X accesses to follow the same pattern results in not having any patterns at all (e.g., when $X = 8$), because this cannot capture the transient interruptions in sequence. In that case, following the major sequential and/or stride trend within a limited page access history window is more resilient to short-term irregularities. Consecutively, when $X = 8$, a majority-based pattern detection can detect 11.3%–29.7% more sequential accesses. Therefore, it can successfully prefetch more accurate pages into the page cache. Besides sequential and stride access patterns, it is also transparent to irregular access patterns; e.g., for Memcached, it can detect 96.4% of the irregularity.

Prefetch Cache Eviction Linux kernel maintains an asynchronous background thread (`kswapd`) to monitor the machine's memory consumption. If the overall memory consumption goes beyond a critical memory pressure or a process's memory usage hits its limit, it determines the eviction candidates by scanning over the in-memory pages to find out the least-recently-used (LRU) ones. Then, it frees up the selected pages from the main memory to allocate new pages. A prefetched cache waits into the LRU list for its turn to get selected for eviction even though it has already been used by a process (Figure 4). Unnecessary pages waiting for eviction in-memory leads to extra scanning time. This extra wait-time due to lazy cache eviction policy adds to the overall latency, especially in a high memory pressure scenario.

3 Remote Memory Prefetching

In this section, we first highlight the characteristics of an ideal prefetcher. Next, we present our proposed online prefetcher along with its different components and the design principles behind them. Finally, we discuss the complexity and correctness of our algorithm.

3.1 Properties of an Ideal Prefetcher

A prefetcher's effectiveness is measured along three axes:

- *Accuracy* refers to the ratio of total cache hits and the total pages added to the cache via prefetching.
- *Coverage* measures the ratio of the total cache hit from the prefetched pages and the total number of requests (e.g., page faults in case of remote memory paging solutions).
- *Timeliness* of an accurately prefetched page is the time gap from when it was prefetched to when it was first hit.

Trade-off An aggressive prefetcher can hide the slower memory access latency by bringing pages well ahead of the access requests. This might increase the accuracy, but as prefetched pages wait longer to get consumed, this wastes the effective cache and I/O bandwidth. On the other hand, a conservative prefetcher has lower prefetch consumption time and reduces cache and bandwidth contention. However, it has lower coverage and cannot hide memory access latency completely. An effective prefetcher must balance all three.

An effective prefetcher must be adaptive to temporal changes in memory access patterns as well. When there is a predictable access pattern, it should bring pages aggressively. In contrast, during irregular accesses, the prefetch rate should be throttled down to avoid cache pollution.

Prefetching algorithms use prior page access information to predict future access patterns. As such, their effectiveness largely depends on how well they can detect patterns and predict. A real-time prefetcher has to face a trade-off between pattern identification accuracy vs. computational complexity and resource overhead. High CPU usage and memory consumption will negatively impact application performance even though they may help in increasing accuracy.

Common Prefetching Techniques The most common and simple form of prefetching is spatial pattern detection [51]. Some specific access patterns (i.e., stride, stream, etc.) can be detected with the help of special hardware (HW) features [33, 35, 66, 80]. However, they are typically applied to identify patterns in instruction access that are more regular; in contrast, data access patterns are more irregular. Special prefetch instructions can also be injected into an application's source code, based on compiler or post-execution based analysis [27,40,41,60,61]. However, compiler-injected prefetching needs a static analysis of the cache miss behavior before the application runs. Hence, they are not adaptive to dynamic cache behavior. Finally, HW- or software (SW)-dependent prefetching techniques are limited to the availability of the special HW/SW features and/or application modification.

Summary An ideal prefetcher should have low computational and memory overhead. It should have high accuracy, coverage, and timeliness to reduce cache pollution; an adaptive prefetch window is imperative to fulfill this requirement. It should also be flexible to both spatial and temporal locality in memory accesses. Finally, HW/SW independence and application transparency make it more generic and robust.

Table 1 compares different prefetching methods.

3.2 Majority Trend-Based Prefetching

Leap has two main components: *detecting trends* and *deter-*

	Low Computational Complexity	Low Memory Overhead	Unmodified Application	HW/SW Independent	Temporal Locality	Spatial Locality	High Prefetch Utilization
Next-N-Line [52]	✓	✓	✓	✓	X	✓	X
Stride [14]	✓	✓	✓	✓	X	✓	X
GHB PC [54]	X	X	✓	X	✓	✓	✓
Instruction Prefetch [27,41]	X	X	X	X	✓	✓	✓
Linux Read-Ahead [72]	✓	✓	✓	✓	✓	✓	X
Leap Prefetcher	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of prefetching techniques based on different objectives.

Algorithm 1 Trend Detection

```
procedure FINDTREND($N_{split}$)
  $H_{size} \leftarrow$ SIZE($AccessHistory$)
  $w \leftarrow H_{size}/N_{split}$        ▷ Start with small detection window
  $\Delta_{maj} \leftarrow \emptyset$
  while true do
    $\Delta_{maj} \leftarrow$ Boyer-Moore on $\{H_{head}, \ldots, H_{head-w-1}\}$
    $w \leftarrow w * 2$
    if $\Delta_{maj} \neq$ major trend then
      $\Delta_{maj} \leftarrow \emptyset$
    if $\Delta_{maj} \neq \emptyset$ or $w > H_{size}$ then
      return $\Delta_{maj}$
  return $\Delta_{maj}$
```

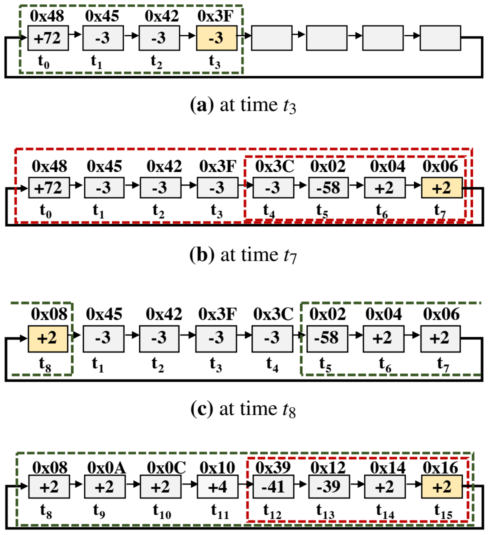
mining what to prefetch. The first component looks for any approximate trend in earlier accesses. Based on the trend availability and prefetch utilization information, the latter component decides how many and which pages to prefetch.

3.2.1 Trend Detection

Existing prefetch solutions rely on strict pattern identification mechanisms (e.g., sequential or stride of fixed size) and fail to ignore temporary irregularities. Instead, we consider a relaxed approach that is robust to short-term irregularities. Specifically, we identify the majority Δ values in a fixed-size (H_{size}) window of remote page accesses (ACCESSHISTORY) and ignore the rest. For a window of size w, a Δ value is said to be the major only if it appears at least $\lfloor w/2 \rfloor + 1$ times within that window. To find the majority Δ, we use the Boyer-Moore majority vote algorithm [17] (Algorithm 1), a linear-time and constant-memory algorithm, over ACCESSHISTORY elements. Given a majority Δ, due to the temporal nature of remote page access events, it can be hypothesized that subsequent Δ values are more likely to be the same as the majority Δ.

Note that if two pages are accessed together, they will be aged and evicted together in the slower memory space at contiguous or nearby addresses. Consequently, the temporal locality in virtual memory accesses will also be observed in the slower page accesses and an approximate stride should be enough to detect that.

Window Management If a memory access sequence follows a regular trend, then the majority Δ is likely to be found in almost any part of that sequence. In that case, a smaller window can be more effective as it reduces the total number of operations. So instead of considering the entire ACCESSHISTORY, we start with a smaller window that starts from the head position (H_{head}) of ACCESSHISTORY. For a window of size w, we find the major Δ appearing in the $H_{head}, H_{head-1}, ..., H_{head-w-1}$ elements.



(a) at time t_3

(b) at time t_7

(c) at time t_8

(d) at time t_{15}

Figure 5: Content of ACCESSHISTORY at different time. Solid colored boxes indicate the head position at time t_i. Dashed boxes indicate detection windows. Here, time rolls over at t_8.

However, in the presence of short-term irregularities, small windows may not detect a majority. To address this, the prefetcher starts with a small detection window and doubles the window size up to ACCESSHISTORY size until it finds a majority; otherwise, it determines the absence of a majority. The smallest window size can be controlled by N_{split}.

Example Let us consider a ACCESSHISTORY with $H_{size} = 8$ and $N_{split} = 2$. Say pages with the following addresses: `0x48`, `0x45`, `0x42`, `0x3F`, `0x3C`, `0x02`, `0x04`, `0x06`, `0x08`, `0x0A`, `0x0C`, `0x10`, `0x39`, `0x12`, `0x14`, `0x16`, were requested in that order. Figure 5 shows the corresponding Δ values stored in ACCESSHISTORY, with t_0 being the earliest and t_{15} being the latest request. At t_i, H_{head} stays at the t_i-th slot.

FINDTREND in Algorithm 1 will initially try to detect a

Algorithm 2 Prefetch Candidate Generation

```
1:  procedure GetPrefetchWindowSize(page $P_t$)
2:    $PW_{size_t}$                          ▷ Current prefetch window size
3:    $PW_{size_{t-1}}$                      ▷ Last prefetch window size
4:    $C_{hit}$                ▷ Prefetched cache hits after last prefetch
5:    if $C_{hit} = 0$ then
6:      if $P_t$ follows the current trend then
7:        $PW_{size_t} \leftarrow 1$              ▷ Prefetch a page along trend
8:      else
9:        $PW_{size_t} \leftarrow 0$                      ▷ Suspend prefetching
10:   else                                   ▷ Earlier prefetches had hits
11:     $PW_{size_t} \leftarrow$ Round up $C_{hit} + 1$ to closest power of 2
12:   $PW_{size_t} \leftarrow \mathbf{min}(PW_{size_t}, PW_{size_{max}})$
13:   if $PW_{size_t} < PW_{size_{t-1}}/2$ then              ▷ Low cache hit
14:     $PW_{size_t} \leftarrow PW_{size_{t-1}}/2$        ▷ Shrink window smoothly
15:   $C_{hits} \leftarrow 0$
16:   $PW_{size_{t-1}} \leftarrow PW_{size_t}$
17:   return $PW_{size_t}$

18: procedure DoPrefetch(page $P_t$)
19:   $PW_{size_t} \leftarrow$ GetPrefetchWindowSize($P_t$)
20:   if $PW_{size_t} \neq 0$ then
21:     $\Delta_{maj} \leftarrow$ FindTrend($N\_split$)
22:     if $\Delta_{maj} \neq \emptyset$ then
23:       Read $PW_{size_t}$ pages with $\Delta_{maj}$ stride from $P_t$
24:     else
25:       Read $PW_{size_t}$ pages around $P_t$ with latest $\Delta_{maj}$
26:   else
27:     Read only page $P_t$
```

trend using a window size of 4. Upon failure, it will look for a trend first within a window size of 8.

At time t_3, FindTrend successfully finds a trend of -3 within the t_0–t_3 window (Figure 5a).

At time t_7, the trend starts to shift from -3 to +2. At that time, t_4–t_7 window does not have a majority Δ, which doubles the window to consider t_0–t_7. This window does not have any majority Δ either (Figure 5b). However, at t_8, a majority Δ of +2 within t_5–t_8 will be adopted as the new trend (Figure 5c).

Similarly, at t_{15}, we have a majority of +2 in the t_8–t_{15}, which will continue to the +2 trend found at t_8 while ignoring the short-term variations at t_{12} and t_{13} (Figure 5d).

3.2.2 Prefetch Candidate Generation

So far we have focused on identifying the presence of a trend. Algorithm 2 determines whether and how to use that trend for prefetching for a request for page P_t.

We determine the prefetch window size (PW_{size_t}) based on the accuracy of prefetches between two consecutive prefetch requests (see GetPrefetchWindowSize). Any cache hit of the prefetched data between two consecutive prefetch requests indicates the overall effectiveness of the prefetch. In case of high effectiveness (i.e., a high cache hit), PW_{size_t} is expanded until it reaches maximum size ($PW_{size_{max}}$). On the other hand, low cache hit indicates low effectiveness; in that case, the prefetch window size gets reduced. However, in the presence of drastic drops, prefetching is not suspended immediately. The prefetch window is shrunk smoothly to make the algorithm flexible to short-term irregularities. When prefetching is suspended, no extra pages are prefetched until a new trend is detected. This is to avoid cache pollution during irregular/unpredictable accesses.

Given a non-zero PW_{size}, the prefetcher brings in PW_{size} pages following the current trend, if any (DoPrefetch). If no majority trend exists, instead of giving up right away, it speculatively brings PW_{size} pages around P_t's offset following the previous trend. This is to ensure that short-term irregularities cannot completely suspend prefetching.

Prefetching in the Presence of Irregularity FindTrend can detect a trend within a window of size w in the presence of at most $\lfloor w/2 \rfloor - 1$ irregularities within it. If the window size is too small or the window has multiple perfectly interleaved threads with different strides, FindTrend will consider it a random pattern. In that case, if the PW_{size} has a non-zero value then it performs a speculative prefetch (line 25) with the previous Δ_{maj}. If that Δ_{maj} is one of the interleaved strides, then this speculation will cause cache hit and continue. Otherwise, PW_{size} will eventually be zero and the prefetcher will stop bringing unnecessary pages. In that case, the prefetcher cannot be worse than the existing prefetch algorithms.

Prefetching During Constrained Bandwidth In Leap, faulted page read and prefetch are done asynchronously. Here, prefetching has a lower priority. In extreme bandwidth constraints, prefetched pages will take a long time to arrive and result in fewer cache hits. This will eventually shrink down PW_{size}. Thus, dynamic prefetch window sizing will help in bandwidth-constrained scenarios.

Effect of Huge Page Linux kernel splits a huge page into 4KB pages before swapping. When transparent huge page is enabled, Leap will be applied on these splitted 4KB pages.

Note that, using huge pages will result in high amplification for dirty data [18]. Besides, average RDMA latencies for 4KB vs 2MB page are 3μs vs 330μs. If huge pages were never split, to maintain single μs latency for 2MB pages, we will need a significantly larger prefetch window size ($PW_{size} \geq 128$), demanding more bandwidth and cache space, and making mispredictions more expensive.

3.3 Analysis

Time Complexity The FindTrend function in Algorithm 1 initially tries to detect trend aggressively within a smaller window using the Boyer-Moor Majority Voting algorithm. If it fails, then it expands the window size. The Boyer-Moor Majority Voting algorithm (line 6) detects a majority element (if any) in $O(w)$ time, where w is the size of the window. In the worst case, it will invoke the Boyer-Moor

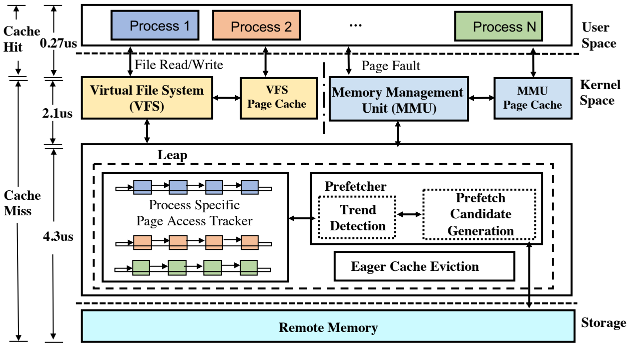


Figure 6: Leap has a faster data path for a cache miss.

Majority Voting algorithm for $O(logH_{size})$ times. However, as the windows are continuous, searching in a new window does not need to start from the beginning and the algorithm never access the same item twice. Hence, the worst-case time complexity of the FindTrend function is $O(H_{size})$, where H_{size} is the size of the AccessHistory queue. For smaller H_{size} the computational complexity is constant. Even for $H_{size} = 32$, the prefetcher provides significant performance gain (§5) that greatly outweighs the slight extra computational cost.

Memory Complexity The Boyer-Moor Majority Voting algorithm operates on constant memory space. FindTrend just invokes the Boyer-Moor Majority Voting algorithm and does not require any additional memory to execute. So, the Trend Detection algorithm needs O(1) space to operate.

Correctness of Trend Detection The correctness of FindTrend depends on that of the Boyer-Moor Majority Voting algorithm, which always provides the majority element, if one exists, in linear time (see [17] for the formal proof).

4 System Design

We have implemented our prefetching algorithm as a data path replacement for memory disaggregation frameworks (we refer to this design as Leap data path) alongside the traditional data path in Linux kernel v4.4.125. Leap has three primary components: a page access tracker to isolate processes, a majority-based prefetching algorithm, and an eager cache eviction mechanism. All of them work together in the kernel space to provide a faster data path. Figure 6 shows the basic architecture of Leap's remote memory access mechanism. It takes only around 400 lines of code to implement the page access tracker, prefetcher, and the eager eviction mechanism.

4.1 Page Access Tracker

Leap isolates each process's page access data paths. The page access tracker monitors page accesses inside the kernel that enables the prefetcher to detect application-specific page access trends. Leap does not monitor in-memory pages (hot pages) because continuously scanning and recording the hardware access bits of a large number of pages causes significant computational overhead and memory consumption. Instead, it monitors only the cache look-ups and records the access sequence of the pages after I/O requests or page faults, trading off a small loss in access pattern detection accuracy for low resource overhead. As temporal locality in the virtual memory space results in a spatial locality in the remote address space, just monitoring the remote page accesses is often enough.

The page access tracker is added as a separate control unit inside the kernel. Upon a page fault, during the page-in operation (`do_swap_page()` under `mm/memory.c`), we notify (`log_access_history()`) Leap's page access tracker about the page fault and the process involved. Leap maintains process-specific fixed-size (H_{size}) FIFO AccessHistory circular queues to record the page access history. Instead of recording exact page addresses, however, we only store the difference between two consecutive requests (Δ). For example, if page faults happen for addresses `0x2`, `0x5`, `0x4`, `0x6`, `0x1`, `0x9`, then AccessHistory will store the corresponding Δ values: 0, +3, -1, +2, -5, +8. This reduces the storage space and computation overhead during trend detection (§3.2.1).

4.2 The Prefetcher

To increase the probability of cache hit, Leap incorporates the majority trend-based prefetching algorithm (§3.2). Here, the prefetcher considers each process's earlier remote page access histories available in the respective AccessHistory to efficiently identify the access behavior of different processes. Because threads of the same process share memory with each other, we choose process-level detection over thread-based. Thread-based pattern detection may result in requesting the same page for prefetch multiple times for different threads.

Two consecutive page access requests are temporally correlated in the sense that they may happen together in the future. The Δ values stored in the AccessHistory records the spatial locality in the temporally correlated page accesses. Therefore, the prefetcher utilizes both temporal and spatial localities of page accesses to predict future page demand.

The prefetcher is added as a separate control unit inside the kernel. While paging-in, instead of going through the default `swapin_readahead()`, we re-route it through the prefetcher's `do_prefetch()` function. Whenever the prefetcher generates the prefetch candidates, Leap bypasses the expensive request scheduling and batching operations of the block layer (`swap_readpage()`/`swap_writepage()` for paging and `generic_file_read()`/`generic_file_write()` for the file systems) and invokes `leap_remote_io_request()` to re-direct the request through Leap's asynchronous remote I/O interface over RDMA (§4.4).

4.3 Eager Cache Eviction

Leap maintains a circular linked list of prefetched caches (PrefetchFifoLruList). Whenever a page is fetched from remote memory, besides the kernel's global LRU lists, Leap adds it at the tail of the linked list. After the prefetch cache gets hit and the page table is updated, Leap marks the page as an eviction candidate. A separate background process continuously removes eviction candidates from PrefetchFifoLruList and frees up those pages to the buddy list. As an accurate prefetcher is timely in using the prefetched data, in Leap, prefetched caches do not wait long to be freed up. For workloads where repeated access to paged-in data is not so common, this eager eviction of prefetched pages reduces the wait time to find and allocate new pages - on average, page allocation time is reduced by 750*ns* (36% less than the usual). Thus, new pages can be brought to the memory more quickly leading to a reduction in the overall data path latency. For workloads where paged-in data is repeatedly used, Leap considers the frequency of access for prefetched pages and exempt them from eager eviction.

However, if the prefetched pages need to be evicted even before they get consumed (e.g., at severe global memory pressure or extreme constrained prefetch cache size scenario), due to the lack of any access history, prefetched pages will follow a FIFO eviction order among themselves from the PrefetchFifoLruList. Reclamation of other memory (file-backed or anonymous page) follows the existing LRU-based eviction technique by `kswapd` in the kernel. We modify the kernel's Memory Management Unit (`mm/swap_state.c`) to add the prefetch eviction related functions.

4.4 Remote I/O Interface

Similar to existing works [10, 32], Leap uses an agent in each host machine to expose a remote I/O interface to the VFS/VMM over RDMA. The host machine's agent communicates to another remote agent with its resource demand and performs remote memory mapping. The whole remote memory space is logically divided into fixed-size memory slabs. A host agent can map slabs across one or more remote machine(s) according to its resource demand, load balancing, and fault tolerance policies.

The host agent maintains a per CPU core RDMA connection to the remote agent. We use the multi-queue IO queuing mechanism where each CPU core is configured with an individual RDMA dispatch queue for staging remote read/write requests. Upon receiving a remote I/O request, the host generates/retrieves a slot identifier, extracts the remote memory address for the page within that slab, and forwards the request to the RDMA dispatch queue to perform read/write over the RDMA NIC. During the whole process, Leap completely bypasses the expensive block layer operations.

Resilience, Scalability, & Load Balancing One can use existing memory disaggregation frameworks [10, 32, 65] with respective scalability and fault tolerance characteristics and still have the performance benefits of Leap. We do not claim any innovation here. In our implementation, the host agent leverages the power of two choices [53] to minimize memory imbalance across remote machines. Remote in-memory replication is the default fault tolerance mechanism in Leap.

5 Evaluation

We evaluate Leap on a 56 Gbps InfiniBand cluster on CloudLab [3]. Our key results are as follows:

- Leap provides a faster data path to remote memory. Latency for 4KB remote page accesses improves by up to 104.04× (24.96×) at the median and 22.06× (17.32×) at the tail in case of Disaggregated VMM (VFS) (§5.1).
- While paging to disk, our prefetcher outperforms its counterparts (Next-K, Stride, and Read-Ahead) by up to 1.62× for cache pollution and up to 10.47× for cache miss. It improves prefetch coverage by up to 37.51% (§5.2).
- Leap improves the end-to-end application completion times of PowerGraph, NumPy, VoltDB, and Memcached by up to 9.84× and their throughput by up to 10.16× over existing memory disaggregation solutions (§5.3).

Methodology We integrate Leap inside the Linux kernel, both in its VMM and VFS data paths. As a result, we evaluate its impact on three primary mediums.

- *Local disks*: Here, Linux swaps to a local HDD and SSD.
- *Disaggregated VMM (D-VMM)*: To evaluate Leap's benefit for disaggregated VMM system, we integrate Leap with the latest commit of Infiniswap on GitHub [4].
- *Disaggregated VFS (D-VFS)*: To evaluate Leap's benefit for a disaggregated VFS system, we add Leap to our implementation of Remote Regions [10], which is not open-source.

For both of the memory disaggregation systems, we use respective load balancing and fault tolerance mechanisms. Unless otherwise specified, we use AccessHistory buffer size $H_{size} = 32$, and maximum prefetch window size $PW_{size_{max}} = 8$.

Each machine in our evaluation has 64 GB of DRAM and 2× Intel Xeon E5-2650v2 with 16 cores (32 hyperthreads).

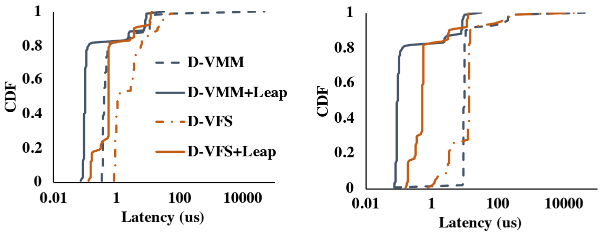
5.1 Microbenchmark

We start by analyzing Leap's latency characteristics with the two simple access patterns described in Section 2.

During sequential access, due to prefetching, 80% of the total page requests hit the cache in the default mechanism. On the other hand, during stride access, all prefetched pages brought in by the Linux prefetcher are unused and every page access request experiences a cache miss.

Due to Leap's faster data path, for **Sequential**, it improves the median by 4.07× and 99^{th} percentile by 5.48× for disaggregated VMM (Figure 7a). For **Stride-10**, as the prefetcher can detect strides efficiently, Leap performs almost as good as it does during the sequential accesses. As a result, in terms of 4KB page access latency, Leap improves disaggregated VMM by 104.04× at the median and 22.06× at the tail (Figure 7b).

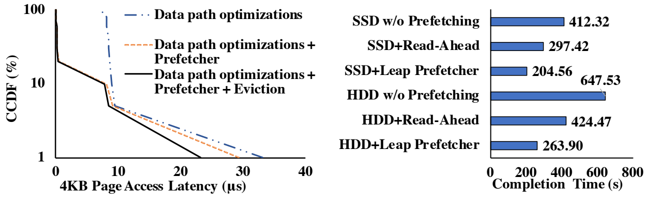
Leap provides similar performance benefits during memory disaggregation through the file abstraction as well. During sequential access, Leap improves 4KB page access latency by 1.99× at the median and 3.42× at the 99^{th} percentile. During



(a) Sequential

(b) Stride-10

Figure 7: Leap provides lower 4KB page access latency for both sequential and stride access patterns.



(a) Benefit Breakdown

(b) Prefetcher with Slow Storage

Figure 8: The prefetcher is effective for different storage systems.

stride access, the median and 99^{th} percentile latency improves by 24.96× and 17.32×, respectively.

Performance Benefit Breakdown For disaggregated VMM (VFS), the prefetcher improves the 99^{th} percentile latency by 25.4% (23.1%) over the optimized data path where Leap's eager cache eviction contributes another 9.7% (8.5%) improvement.

As the idea of using far/remote memory for storing cold data is getting more popular these days [9,32,45], throughout the rest of the evaluation, we focus only on remote paging through a disaggregated VMM system.

5.2 Performance Benefit of the Prefetcher

Here, we focus on the effectiveness of the prefetcher itself. We use four real-world memory-intensive applications and workload combinations (Figure 3) used in prior works [10,32].

- TunkRank [8] on PowerGraph [29] to measure the influence of a Twitter user from the follower graph [44]. This workload has a significant amount of stride, sequential, and random access patterns.
- Matrix multiplication on NumPy [57] over matrices of floating points. This has mostly sequential patterns.
- TPC-C benchmark [7] on VoltDB [70] to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. This has mostly random with a few amount of sequential patterns.
- Facebook's ETC workload [13] on Memcached [5]. We use 10 million SET operations to populate the Memcached server. Then we perform another 10 million queries (5%SETs, 95%GETs). This has mostly random patterns.

The peak memory usage of these applications varies from 9–38.2 GB. To prompt remote paging, we limit an application's memory usage through `cgroups` [2]. To separate the benefit of the prefetcher, we run all of the applications on disk (with existing block layer-based data path) with 50% memory limit.

5.2.1 Prefetch Utilization

We observe the benefit of Leap's prefetcher over following practical and realtime prefetching techniques:

- *Next-N-Line Prefetcher* [52] aggressively brings N pages sequentially mapped to the page with the cache miss if they are not in the cache.
- *Stride Prefetcher* [14] brings pages following a stride pattern relative to the current page upon a cache miss. The aggressiveness of this prefetcher depends on the accuracy of the past prefetch.
- *Linux Read-Ahead* prefetches an aligned block of pages containing the faulted page [72]. Linux uses prefetch hit count and an access history of size 2 to control the aggressiveness of the prefetcher.

Impact on the Cache As the volume of data fetched into the cache increases, the prefetch hit rate increases as well. However, thrashing begins as soon as the working set exceeds the cache capacity. As a result, useful demand-fetched pages are evicted. Table 2 shows that Leap's prefetcher uses fewer page caches (4.37–62.13%) than the other prefetchers for every workload.

A successful prefetcher reduces the number of cache misses by bringing the most accurate pages into the cache. Leap's prefetcher experiences fewer cache miss events (1.1–10.47×) and enhances the effective usage of the cache space.

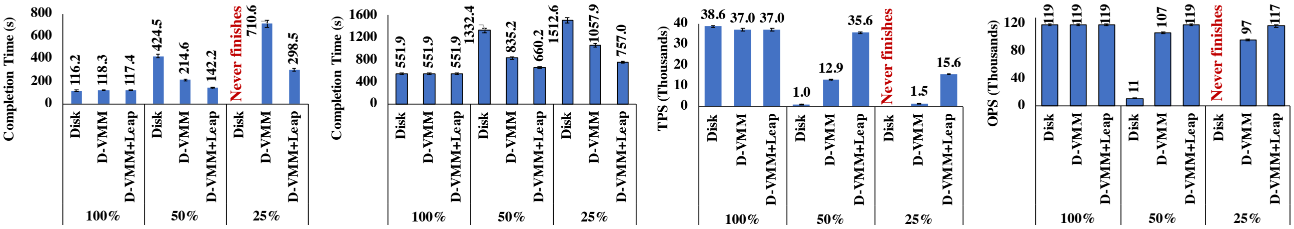
Application Performance Due to the improvement in cache pollution and reduction of cache miss, using Leap's prefetcher, all of the applications experience the lowest completion time. Based on the access pattern, Leap's prefetcher improves the application completion time by 7.4–75.3% over Linux's default Read-Ahead prefetching technique (Table 2).

Effectiveness If a prefetcher brings every possible page in the page cache, then it will be 100% accurate. However, in reality, one cannot have an infinite cache space due to large data volumes and/or multiple applications running on the same machine. Besides, optimistically bringing pages may create cache contention, which reduces the overall performance.

Leap's prefetcher trades off cache pollution with comparatively lower accuracy. In comparison to other prefetchers, it shows 0.3–10.8% lower accuracy (Table 2). This accuracy loss is linear to the number of cache adds done by the prefetchers. Because the rest of the prefetchers bring in too many pages, their chances of getting lucky hits increase too. Although Leap has the lowest accuracy, its high coverage (0.7–37.5%) allows it to serve with accurate prefetches with a lower cache pollution cost. At the same time, it has an im-

	PowerGraph				NumPy				VoltDB				Memcached			
	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap	Next-N-Line	Stride	Read-Ahead	Leap
Cache Add (millions)	4.88	3.88	3.85	**3.01**	10.75	10.52	10.61	**10.08**	6.50	6.23	5.91	**5.20**	4.65	4.14	4.06	**3.25**
Cache Miss (millions)	1.11	1.61	0.26	**0.15**	0.13	0.16	0.14	**0.12**	1.53	2.24	0.96	**0.90**	1.44	1.39	1.36	**0.96**
Completion Time (s)	683.92	885.86	462.54	**263.90**	1410.30	1380.10	1332.40	**1240.60**	2017.47	2454.72	2064.60	**1799.84**	382.54	374.60	366.91	**302.43**
Accuracy (%)	**55.30**	45.60	45.10	44.60	**89.60**	89.40	89.20	88.90	**40.20**	39.50	39.90	37.60	41.80	**42.10**	41.90	39.40
Coverage (%)	70.90	52.30	86.80	**89.80**	95.80	96.30	96.80	**98.60**	61.20	47.40	68.50	**71.00**	51.70	52.40	56.90	**57.60**
Timeliness (ms) - 95^{th} Percentile	19.10	**0.03**	0.39	0.07	10.34	**0.02**	0.24	0.06	22125.14	**34.32**	64314.96	776.68	32417.89	**466.64**	46679.77	886.67

Table 2: Leap's prefetcher reduces cache pollution and cache miss events. With higher coverage, better timeliness and almost similar accuracy, the prefetcher outperforms its counterparts in terms of application level performance. Here, shaded numbers indicate the best performances.



(a) PowerGraph Completion Time **(b)** NumPy Completion Time **(c)** VoltDB Throughput **(d)** Memcached Throughput

Figure 9: Leap provides lower completion times and higher throughput over Infiniswap's default data path for different memory limits. Note that lower is better for completion time, while higher is better for throughput. Disk refers to HDD in this figure.

proved timeliness over Read-Ahead (4–52.6×) at the 95^{th} percentile. Due to the higher coverage, better timeliness, and almost similar accuracy, Leap's prefetcher thus outperforms others in terms of application-level performance. Note that despite having the best timeliness, Stride has the worst coverage and completion time that impedes its overall performance.

5.2.2 Performance Benefit Breakdown

Figure 8a shows the performance benefit breakdown for each of the components of Leap's data path. For PowerGraph at 50% memory limit, due to data path optimizations, Leap provides with single μs latency for 4KB page accesses up to the 95^{th} percentile. Inclusion of the prefetcher ensures sub-μs 4KB page access latency up to the 85^{th} percentile and improves the 99^{th} percentile latency by 11.4% over Leap's optimized data path. The eager eviction policy reduces the page cache allocation time and improves the tail latency by another 22.2%.

5.2.3 Performance Benefit for HDD and SSD

To observe the usefulness of the prefetcher for different slow storage systems, we incorporate it into Linux's default data path while paging to SSD. For PowerGraph, Leap's prefetcher improves the overall application run time by 1.45× (1.61×) for SSD (HDD) over Linux's default prefetcher (Figure 8b).

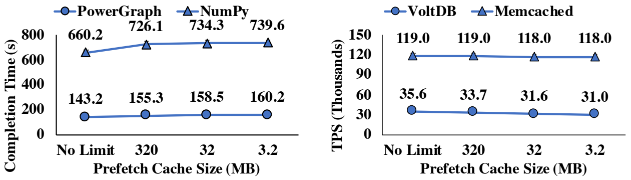
5.3 Leap's Overall Impact on Applications

Finally, we evaluate the overall benefit of Leap (including all of its components) for the applications mentioned in Section 5.2. We limit an application's memory usage to fit 100%, 50%, 25% of its peak memory usage. Here, we considered the extreme memory constrain (e.g., 25%) to validate the applicability of Leap to recent resource (memory) disaggregation frameworks that operate on a minimal amount of local memory [65].

PowerGraph PowerGraph suffers significantly for cache misses in Infiniswap (Figure 9a). In contrast, Leap increases the cache hit rate by detecting 19.03% more remote page access patterns over Read-Ahead. The faster the prefetch cache hit happens, the faster the eager cache eviction mechanism frees up page caches and eventually helps in faster page allocations for a new prefetch. Besides, due to more accurate prefetching, Leap reduces the wastage in both cache space and RDMA bandwidth. This improves 4KB remote page access time by 8.17× and 2.19× at the 99^{th} percentile for 50% and 25% cases, respectively. Overall, the integration of Leap to Infiniswap improves the completion time by 1.56× and 2.38× at 50% and 25% cases, respectively.

NumPy Leap can detect most of the remote page access patterns (10.4% better than Linux's default prefetcher). As a result, similar to PowerGraph, for NumPy, Leap improves the completion time by 1.27× and 1.4× for Infiniswap at 50% and 25% memory limit, respectively (Figure 9b). The 4KB page access time improves by 5.28× and 2.88× at the 99^{th} percentile at 50% and 25% cases, respectively.

VoltDB Latency-sensitive applications like VoltDB suffer significantly due to paging. During paging, due to Linux's slower data path, Infiniswap suffers 65.12% and 95.72% lower throughput than local memory behavior at 50% and 25% cases, respectively. In contrast, Leap's better prefetching (11.6% better than Read-Ahead) and instant cache eviction improves the 4KB page access time – 2.51× and 2.7× better 99^{th} percentile at 50% and 25% cases, respectively. However, while executing short random transactions, VoltDB has irregular page access patterns (69% of the total remote page



(**a**) Completion Time (**b**) Throughput

Figure 10: Leap has minimal performance drop for Infiniswap even in the presence of O(1) MB cache size.

accesses). At that time, our prefetcher's adaptive throttling helps the most by not congesting the RDMA. Overall, Leap faces smaller throughput loss (3.78% and 57.97% lower than local memory behavior at 50% and 25% cases, respectively). Leap improves Infiniswap's throughput by 2.76× and 10.16× at 50% and 25% cases, respectively (Figure 9c).

Memcached This workload has a mostly random remote page access pattern. Leap's prefetcher can detect most of them and avoids prefetching in the presence of randomness. This results in fewer remote requests and less cache pollution. As a result, Leap provides Memcached with almost the local memory level behavior at 50% memory limit while the default data path of Infiniswap faces 10.1% throughput loss (Figure 9d). At 25% memory limit, Leap deviates from the local memory throughput behavior by only 1.7%. Here, the default data path of Infiniswap faces 18.49% throughput loss. In this phase, Leap improves Infiniswap's throughput by 1.11× and 1.21× at 50% and 25% memory limits, respectively. Here, Leap provides with 5.94× and 1.08× better 99th percentile 4KB page access time at 50% and 25% cases, respectively.

Performance Under Constrained Cache Size To observe Leap's performance benefit in the presence of limited cache size, we run the four applications in 50% memory limit configuration at different cache limits (Figure 10).

For Memcached, as most of the accesses are of random patterns, most of the performance benefit comes from Leap's faster slow path. For the rest of the applications, as the prefetcher has better timeliness, most of the prefetched caches get used and evicted before the cache size hits the limit. Hence, during O(1) MB cache size, all of these applications face minimal performance drop (11.87–13.05%) compared to the unlimited cache space scenario. Note that, for NumPy, 3.2 MB cache size is only 0.02% of its total remote memory usage.

Multiple Applications Running Together We run all four applications on a single host machine simultaneously with their 50% memory limit and observe the performance benefit of Leap for Infiniswap when multiple throughput- (PowerGraph, NumPy) and latency-sensitive applications (VoltDB, Memcached) concurrently request for remote memory access (Figure 11). As Leap isolates each application's page access path, its prefetcher can consider individual access patterns while making prefetch decisions. Therefore, it brings more accurate remote pages for each of the applications and reduces the contention over the network. As a result, overall application-level performance improves by 1.1–2.4× over Infiniswap. To enable aggregate performance comparison, here, we present the end-to-end completion time of application-workload combinations defined earlier; application-specific metrics improve as well.

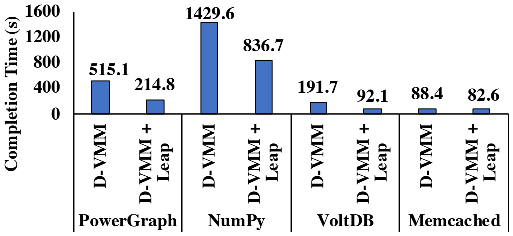


Figure 11: Leap improves application-level performance when all four applications access remote memory concurrently.

6 Discussion and Future Work

Thread-specific Prefetching Linux kernels today manage memory address space at the process level. Thread-specific page access tracking requires a significant change in the whole virtual memory subsystem. However, this would help efficiently identify multiple concurrent streams from different threads. Low-overhead, thread-specific page access tracking and prefetching can be an interesting research direction.

Concurrent Disk and Remote I/O Leap's prefetcher can be used for both disaggregated and existing Linux Kernels. Currently, Leap runs as a single memory management module on the host server where paging is allowed through either existing block layers or Leap's remote memory data path. The current implementation does not allow the concurrent use of both block layer and remote memory. Exploring this direction can lead to further benefits for systems using Leap.

Optimized Remote I/O Interface In this work, we focused on augmenting existing memory disaggregation frameworks with a leaner and efficient data path. This allowed us to keep Leap transparent to the remote I/O interface. We believe that exploring the effects of load balancing, fault-tolerance, data locality, and application-specific isolation in remote memory as well as an optimized remote I/O interface are all potential future research directions.

7 Related Work

Remote Memory Solutions A large number of software systems have been proposed over the years to access remote machine's memory for paging [1, 21, 23, 26, 32, 45, 46, 50, 55, 64, 65], global virtual machine abstraction [6, 25, 43], and distributed data stores and file systems [10, 22, 42, 47, 58]. Hardware-based remote access using PCIe interconnects [48] or extended NUMA memory fabric [56] are also proposed to disaggregate memory. Leap is complementary to these works.

Kernel Data Path Optimizations With the emergence of faster storage devices, several optimization techniques, and design principles have been proposed to fully utilize faster hardware. Considering the overhead of the block layer, different service level optimizations and system re-designs have been proposed – examples include parallelism in batching and queuing mechanism [16, 75], avoiding interrupts and context switching during I/O scheduling [12, 20, 74, 76], better buffer cache management [34], etc. During remote memory access, optimization in data path has been proposed through request batching [37, 38, 71], eliminating page migration bottleneck [73], reducing remote I/O bandwidth through compression [45], and network-level block devices [46]. Leap's data path optimizations are inspired by many of them.

Prefetching Algorithms Many prefetching techniques exist to utilize hardware features [33, 35, 66, 80], compiler-injected instructions [27, 40, 41, 60, 61], and memory-side access pattern [24, 54, 67–69] for cache line prefetching. They are often limited to specific access patterns, application behavior, or require specified hardware design. More importantly, they are designed for a lower level memory stack.

A large number of entirely kernel-based prefetching techniques have also been proposed to hide the latency overhead of file accesses and page faults [19, 24, 31, 39, 72]. Among them, Linux Read-Ahead [72] is the most widely used. However, it does not consider the access history to make prefetch decisions. It was also designed for hiding disk seek time. Therefore, its optimistic looking around approach often results in lower cache utilization for remote memory access.

To the best of our knowledge, Leap is the first to consider a fully software-based, kernel-level prefetching technique for DRAM with remote memory as a backing storage over fast RDMA-capable networks.

8 Conclusion

The paper presents Leap, a remote page prefetching algorithm that relies on majority-based pattern detection instead of strict detection. As a result, Leap is resilient to short-term irregularities in page access patterns of multi-threaded applications. We implement Leap in a leaner and faster data path in the Linux kernel for remote memory access over RDMA without any application or hardware modifications.

Our integrations of Leap with two major memory disaggregation systems (namely, Infiniswap and Remote Regions) show that the median and tail remote page access latencies improves by up to 104.04× and 22.62×, respectively, over the state-of-the-art. This, in turn, leads to application-level performance improvements of 1.27–10.16×. Finally, Leap's benefits extend beyond disaggregated memory – applying it to HDD and SSD leads to considerable performance benefits as well.

Leap is available at https://github.com/SymbioticLab/leap.

Acknowledgments

We want to thank the anonymous reviewers, our shepherd, Vincent Liu, and SymbioticLab members for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants CNS-1845853, CCF-1629397, and CNS-1617773.

References

[1] Accelio based network block device. https://github.com/accelio/NBDX.

[2] cgroups. https://wiki.archlinux.org/index.php/cgroups.

[3] CloudLab. https://www.cloudlab.us.

[4] Infiniswap github repository. https://github.com/SymbioticLab/infiniswap.

[5] Memcached - A distributed memory object caching system. http://memcached.org.

[6] The versatile SMP (vSMP) architecture. http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[7] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc.

[8] A twitter analog to PageRank. http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

[9] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ASPLOS*, 2017.

[10] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *ATC*, 2018.

[11] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[12] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *HotStorage*, 2011.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 2012.

[14] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ACM/IEEE Conference on Supercomputing*, 1991.

[15] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.

[16] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *SYSTOR*, 2013.

[17] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning*. 1991.

[18] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: FPGA acceleration for remote memory. In *HotOS*, 2019.

[19] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *OSDI*, 1994.

[20] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, 2010.

[21] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[22] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[23] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[24] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy. Speculative paging for future NVM storage. In *MEMSYS*, 2017.

[25] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[26] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical report, University of Washington, 1991.

[27] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *MICRO*, 2011.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[31] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *USTC*, 1994.

[32] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[33] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[34] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *FAST*, 2005.

[35] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[36] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[37] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *ATC*, 2016.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[39] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. *SIGPLAN Not.*, 2002.

[40] M. Khan, A. Sandberg, and E. Hagersten. A case for resource efficient prefetching in multicores. In *ICPP*, 2014.

[41] A. Kolli, A. Saidi, and T. F. Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *MICRO*, 2013.

[42] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[43] Y. Kuperman, J. Nider, A. Gordon, and D. Tsafrir. Paravirtual Remote I/O. In *ASPLOS*, 2016.

[44] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[45] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[46] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[47] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[48] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[49] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.

[50] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *ATC*, 1996.

[51] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 1984.

[52] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 2016.

[53] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, 2001.

[54] K. Nesbit and J. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 2005.

[55] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[56] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[57] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006.

[58] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[59] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[60] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, 2015.

[61] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS*, 2004.

[62] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[63] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. In *PVLDB*, 2015.

[64] A. Samih, R. Wang, C. Maciocco, T.-Y. C. Tai, R. Duan, J. Duan, and Y. Solihin. Evaluating dynamics and bottlenecks of memory collaboration in cluster systems. In *CCGrid*, 2012.

[65] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[66] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *MICRO*, 2000.

[67] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[68] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA*, 2009.

[69] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[70] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 2013.

[71] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *SOSP*, 2017.

[72] Y. Wiseman, S. Jiang, Y. Wiseman, and S. Jiang. *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. Information Science Reference - Imprint of: IGI Publishing, 2009.

[73] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2019.

[74] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *FAST*, 2012.

[75] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-level i/o scheduling. In *SOSP*, 2015.

[76] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Comput. Syst.*, 2014.

[77] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *PVLDB*, 2017.

[78] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[79] Y. Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance Isolation Anomalies in RDMA. In *KBNets*, 2017.

[80] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: Improving timeliness in data prefetching. In *ICS*, 2010.

go-pmem: Native Support for Programming Persistent Memory in Go

Jerrin Shaji George* *VMware*

Mohit Verma* *VMware*

Rajesh Venkatasubramanian *VMware*

Pratap Subrahmanyam *VMware*

Abstract

Persistent memory offers persistence and byte-level addressability at DRAM-like speed. Operating system support and some user-level library support for persistent memory programming has emerged. But we think lack of native programming language support is an impediment to a programmer's productivity.

This paper contributes *go-pmem*, an open-source extension to the Go language compiler and runtime that natively supports programming persistent memory. *go-pmem* extends *Go* to introduce a runtime garbage collected persistent heap. Often persistent data needs to be updated in a transactional (i.e., crash consistent) manner. To express transaction boundaries, *go-pmem* introduces a new *txn* block which can include most *Go* statements and function calls. *go-pmem* compiler uses static type analysis to log persistent updates and avoid logging volatile variable updates whenever possible.

To guide our design and validate our work, we developed a feature-poor Redis server *go-redis-pmem* using *go-pmem*. We show that *go-redis-pmem* offers more than 5x throughput than unmodified Redis using a high-end NVMe SSD on memtier benchmark and can restart up to 20x faster than unmodified Redis after a crash. In addition, using compiler microbenchmarks, we show *go-pmem*'s persistent memory allocator performs up to 40x better and transactions up to 4x faster than commercial libraries like PMDK and previous work like Mnemosyne.

1 Introduction

At present, there is a large gap between the access latency for memory and storage hierarchy. The fastest tier of storage, e.g., SSDs, which provide persistence, can be 100x-1000x slower than DRAM, which are volatile and lose all their data on a power cycle. Applications carefully place their working set data in DRAM where the access latency is between 50-100ns, and every so often, perhaps under the guidance of the user via policies, save the data from DRAM to the storage tier. Over the past three decades, applications and operating systems have evolved to orchestrate this data movement in a highly efficient way.

Persistent memory (*pmem*) is a new type of random-access memory that offers persistence and byte-level addressability at DRAM-like access speed [19]. Intel© Optane™ DC Persistent Memory [12] is an example of a readily available persistent memory product. Persistent memory is now becoming increasingly available in servers [1]. Operating systems such as Linux have had support to use pmem as faster storage disks for some time now [6].

The obvious way to consume pmem is as a faster storage device (block mode access) by running an unchanged application on top of a file system. We demonstrate a significant improvement in the throughput of Redis when it is run on top of Linux ext4 filesystem using persistent memory as the block storage device. But, we were also able to show even further performance improvements with a Redis that was hand modified to use byte-addressable persistent memory for its in-memory database. Unsurprisingly, we were also able to show that a good bit of the application's I/O processing code that store the data from volatile DRAM to storage, can now be retired as data is being immediately persisted all over the application. So, using the byte-addressable mode of pmem delivers more performance than using it in block mode, and also lowers the overall complexity in the application code. We delve deeper into this in §2.

We argue that databases like Redis are fast and popular because they highly optimize the data structures used in volatile memory. However, applications must write these data structures into serial buffers for persistence and these optimizations are lost. With byte-addressable persistent memory, applications can directly persist and retrieve their data structures without serialization. And so, the main focus of this work is that we strive to make manipulating persistent memory similar to manipulating volatile memory.

Towards this, we contribute *go-pmem*, an open source extension to the popular Go programming language. go-pmem provides a familiar Go-esque programming model to the appli-

*Co-first authors ordered alphabetically cation developers to use persistent memory at byte-level granularity. We arrived at our current model by implementing a feature-poor implementation of Redis (*go-redis-pmem*) (§5.3) that directly uses byte-addressable pmem. With go-pmem, we contribute a programming model with the following design goals:

1. Single type system. No separate persistent types as in [18, 32].
2. Pointers remain unchanged, i.e., no fat pointers. A consequence of this is we implement pointer swizzling (§4.3.2).
3. Support two heaps, volatile and persistent: go-pmem extends Go runtime to manage pmem and uses pointers to identify objects in pmem.
4. Allow pointers both across and within the persistent and volatile heap, but make system safe across crash and recovery. To do this, go-pmem extends Go's garbage collector (GC) to work across both the heaps.
5. After restart, allow applications to retrieve back data stored in pmem by associating it with a string name. These are called named-objects (§4.4.1).
6. Allow transactional code blocks by requiring the user to demarcate these with a new Go *txn* keyword.
7. Reuse functions to operate on data in volatile or persistent memory.
8. Allow allocation and update of pmem resident data structures outside a transaction as long as they are not reachable from a *named object*. In case of a crash-and-recovery, go-pmem's GC will garbage collect such objects thus avoiding any persistent memory leaks.

In a set of microbenchmarks, we see that go-pmem performs up to 40x better than other pmem libraries languages. go-redis-pmem offers 5x more throughput than unmodified Redis on an SSD against memtier benchmark and restarts up to 20x faster than unmodified Redis. We explain our evaluation methodologies more in §6. To the best of our knowledge, ours is the first expansive effort to change Go to support persistent memory. In §5, we explain how Go's existing design features helped and challenged us towards our idea of supporting pmem.

go-pmem is developed from the Go 1.11 code base, and is fully open-sourced. Links to the respective repositories can be found at https://vmware.github.io/persistent-memory-projects/.

2 Background

To give more context into our work, we begin by asking why should anyone care about persistent memory?

2.1 Experiments with Redis

Over the years, a lot of work has gone into optimizing data intensive applications. To reduce the latency of data access, these applications keep frequently used data in DRAM, and flush the dirty data from DRAM to disks/SSD at certain well-defined points. For example, Redis [24] allows users to persist their data to disks/SSD in an append-only-file (AOF). But if no persistence mode is used, a crash at any point would cause all the in-memory data to be lost. If the user cannot afford any data loss at all, he has to write to the AOF file after each write request and as such sees significantly reduced throughput.

In all the experiments in this section, we run Redis in the zero data loss mode, as that is the fair way to compare with byte addressable persistent memory which provides zero data loss. To see the benefits of persistent memory, we now conduct the following experiment:

1. Use the memtier benchmark as the load generating input workload [25], configured to issue read-write requests in a 70:30 ratio.
2. Run unmodified Redis-3.2, in the following two modes:
   (a) saving the AOF on an SSD and
   (b) saving the AOF on a persistent memory device.
3. Run Redis-3.2 hand modified for byte-addressability taken from [11].

Figure 1 shows that just by running unchanged Redis on a pmem device as block storage, the throughput increases by up to 4x. This configuration gives a higher throughput owing to the inherent faster access time of persistent memory. Figure 1's *Pmem Block IO* curve confirms this. The third curve (PMDK-Redis) [11] shows the performance of a Redis server modified to write data to persistent memory used in byte-addressable mode. In fact, PMDK-Redis even outperforms Redis running on pmem as block-storage.

To understand this, we perform another experiment. We run unmodified Redis-3.2 once with AOF disabled and once with always-fsync-AOF option on *nullfsvfs* virtual file system [2]. This file system treats all read and write system calls to the storage media as no-ops. So the difference in performance is entirely due to serializing data in the application and the overheads in making system calls. This is the reason why in Fig. 2 Redis with AOF enabled on nullfsvfs is slower than Redis with AOF disabled. Because pmem is faster than SSD, the time spent accessing the device is significantly reduced and the overhead of the software stack become a significant portion of the overall application latency.

These two experiments convince us that using persistent memory can give significant throughput improvements in data intensive applications and that the most efficient way for applications to access persistent memory is through direct CPU load/store instructions bypassing any file-system/PCIe overheads. A more extensive study on persistent memory can be found in [37]. Since memory is accessed in 64-byte cache lines, the CPU reads only what it needs to read, instead of rounding every access up to a block size, like storage. Linux also allows persistent memory to be used in a direct-access (DAX) mode [6, 20, 21]. This mode allows users to mmap files in persistent memory into their virtual address space and access it through loads/stores, bypassing the OS page cache

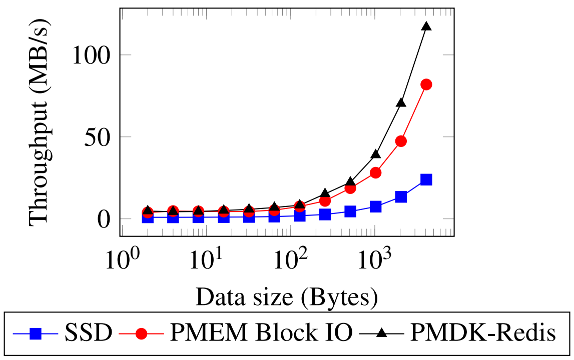


Figure 1: Redis throughput comparison against memtier

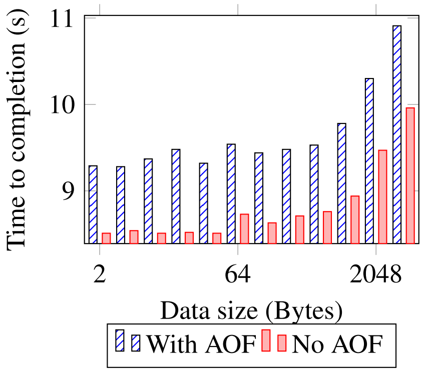


Figure 2: Redis-server runtime on nullfsvfs

and file systems. In the rest of this work, we use pmem in *byte-addressable* mode, unless we specifically mention block storage mode.

2.2 Why Change a Programming Language?

Hand porting applications to use byte-addressable pmem can introduce fragility in the code. Just think of a situation, where the developer misses to guard a single store to persistent memory within a transaction - the bug will be very difficult to discover. So, the next question is: how can the porting process be made easier? Ideally, any acceptable solution should have the following features:

1. Be similar to existing programming models.
2. Work transparently for systems not supporting persistent memory. I.e., we want to be able to write functions that can operate on data in persistent memory or volatile memory.
3. Fast execution time.

One approach is via ad hoc libraries. As we will discuss in §3 and §4, these libraries expose a programming model different than existing programming models for volatile memory. In particular, memory management becomes tricky and these solutions either don't provide a simple and complete programming model, or go through complicated steps to keep it simple. We argue that programming languages already manage volatile memory. So, persistent memory which is a special type of memory and is also byte-addressable, should also be managed by programming languages.

Such a language should at least provide:

1. Persistent memory allocations on heap
2. Garbage collection of persistent heap objects
3. Support modifying persistent memory in a crash-consistent way
4. Support recovery from crashes, i.e., include support for reverting inconsistent updates.
5. Not require offline processing of persistent data regions
6. Similar to existing programming models, for easier adoption.

2.3 Why Go?

We chose Go because it is a high-level managed language with an easily extendable runtime. Moreover, we use some of Go's design to our advantage. For example, we reuse Go's mark-and-sweep garbage collector to garbage collect pmem. Go uses static escape analysis to determine the scope of objects at compile time and intelligently places objects on the stack or the heap. We extend this to track accesses to volatile and persistent heap and prevent unnecessary pmem accesses. Additionally, the potential benefits of high-level languages are well understood. Automatic memory management in HLLs like Go reduce programmer effort and use-after-free bugs. These kinds of bugs are more dangerous with the use of pmem because any memory leak will survive crashes/restarts. Go's type-safety helped us to avoid special data types and invalid memory accesses. Go also has an active developer community with increasing adoption in systems community (e.g. popular software such as Kubernetes, Docker etc. are written in Go) and this fits well with our plan to open-source our changes.

3 Related Work

Most of the previous efforts to program persistent memory fall into two categories:

1. Ad hoc library to support pmem. This library usually allows special objects to be created/updated/destroyed in a crash-consistent way through special APIs [17, 18, 30].
2. Programming language enhancements to manage pmem and instrument user code with transactions [26, 27, 32, 45, 46].

In section 6, we compare go-pmem's performance with works in both categories, PMDK [18] (a library) and Mnemosyne [45], Makalu [26] (language changes).

Previous efforts often provide a new allocator for pmem and require the user to free memory allocated in pmem [7, 18, 26]. Either they don't handle leaks in persistent memory [45] or use special objects and data types to maintain reference counts [26, 30, 32] for garbage collection. Often they provide offline

Library (language)	Model	References	Transactions			Heap	
			Semantics	Implementation	Fn calls	Growth	Reloc.
go-pmem (Go)	Compiler support, Library	Direct pointers	Explicit - user tagged	Undo logs	Yes	Yes	Yes
PMDK [18] (C,C++)	Library	Fat pointers	Explicit - user tagged	Undo logs	Yes	No	Yes
Mnemosyne [45] (C)	Compiler support, Library	Direct pointers	Explicit - user tagged	Redo logs	No	No	No
Makalu [26] (C)	Library	Direct pointers	N/A	N/A	N/A	No	No
Atlas [27] (C)	Compiler support, Library	Direct pointers	Implicit - using locks	Undo logs	No	No	No
Autopersist [43] (Java)	Compiler, JVM support	Direct references	Explicit - user tagged Implicit - durable roots	Undo logs	Yes	No	No
Espresso [46] (Java)	Compiler, JVM support	Direct references	Explicit - user tagged	Persistent objects	Yes	No	No
iDO [38] (C,C++)	Compiler support	Direct pointers	Implicit - using locks	Atomicity via resumption	No	No	No
JUSTDO [36] (C,C++)	Library	Direct pointers	Implicit - using locks	Atomicity via resumption	No	No	No

Table 1: Table comparing features of various pmem libraries

tools to check pmem file for garbage [18, 26]. Some require that users must do pmem memory allocations within transactions [18]. To access data in pmem they usually provide fat pointers [18, 30] or allow direct pointers and ignore pointer swizzling [26, 46]. We did not find any previous work that handled growing persistent heap at runtime. In section 6.4.2 we discuss why this is important.

[31, 45] also provide concurrency through transactions. Of course, transactions for programming languages has been a well-researched topic [34]. However, we don't pursue transactions as a way of solving concurrency, but more from a point of view of crash-consistent/atomic updates while minimally getting in the way of Go's existing concurrency paradigms. In this regard, we are similar to [32, 43] but are more flexible (see section 4.5).

Similar to [27, 32] we use a modular SSA pass to inject undo transaction logging statements to user code. But unlike [32] we do not ask the user to provide special pragmas for faster execution. We believe these optimizations complicate the programming model. We also allow function calls within transactions unlike [45] and want the functions to be reused for persistent and volatile data.

Another design point is how to demarcate the transactional code block. For example, Atlas [27] uses existing locks and constructs a happens-before graph to determine order of logs and updates. Autopersist [43] on the other hand, is quite different. Like go-pmem, they allow users to explicitly demarcate transactional code blocks. But, they also allow users to write normal Java code and when they point to a volatile data from a pmem root, Java runtime moves the transitive closure of the volatile data to pmem transactionally. go-pmem allows updates to pmem-resident objects outside transactions as long as they are not pointed to by a named object.

So far, most of these efforts have been confined to C, C++ with some work in Java [17, 43, 46] and OpenJDK [15]. There have been calls for these programming languages to support persistent memory [4] but we don't know of any concrete changes yet. Table 1 presents a concise summary comparing the features of various pmem libraries. *Fn calls* captures whether the transaction semantics allows function calls inside a transaction. *Heap Growth* indicates whether the library supports a growable heap design and *Heap Reloc.* states if the library supports relocating the pmem heap on an application restart.

4 Design

The design of go-pmem is driven mainly by two considerations:

1. The changes should be accepted to the Go language. This translated to reusing existing Go compiler techniques and keeping our changes to a minimum.
2. Provide a familiar programming model to developers

that has a minimal interface and is congruent with the existing Go infrastructure. This would help in easier acceptance and adoption by the Go community.

4.1 Programming Model

Listing 1 shows how to add a new node to a linkedlist resident in pmem using go-pmem. We have highlighted how this is different than normal Go code adding a node to a linkedlist in volatile memory. Memory in pmem is allocated using *pnew*, similar to Go's *new* and all updates are made transactional using codeblock *txn("undo")*. We argue that this is very similar to how Go code is written today. How we managed to get a programming model like Listing 1 is discussed in the rest of this section.

```
package main
import "pmem"            // <-
import "transaction"     // <-

// add new node to tail; return new tail
func addNode(tail *node) *node {
  n := pnew(node)        // <-
  txn("undo") {          // <-
    mutex.Lock()
    n.prev = tail
    updateTail(tail, n)
    mutex.Unlock()
  }                      // <-
  return n
}

func updateTail(tail, n *node) {
  txn("undo") {          // <-
    tail.next = n
  }                      // <-
}
```

Listing 1: Add node to a linkedlist in pmem

4.2 Language Constructs

We have added two new APIs to Go semantics to support persistent memory allocations:

```
func pnew(Type) *Type
func pmake(t Type, size ...IntType) Type
```

Just like new [8], *pnew* creates a zero-value object of the Type argument in pmem and returns a pointer to this object. The *pmake* API is used to create a slice in pmem. The semantics of pmake is the same as the make API in Go.

4.3 Runtime Design

Go runtime uses datastructures such as mcache, mspan, mcentral, mheap, etc. to store the metadata related to the heap. A span is a contiguous region in memory (one or more pages) from which the allocator allocates similar-sized objects. mspan stores metadata about a span. mcache is used to cache spans at a thread level. If an allocation request can be satisfied using the cached span, then it can be done so without acquiring any locks, making such allocations very fast. If not, a new span is obtained from the mcentral or mheap. mcentral is a central store of small spans (object size <= 32K) and mheap is a central store of freed spans and large spans.

The heap is managed in arenas of size 64MB. Each arena data-structure stores the following metadata:

1. Span table - Span table is an array that holds a reference to the mspan object corresponding to that virtual page index.
2. Heap type bitmap - The heap type bits are used by the GC to identify what regions in memory have pointers in them. The GC uses these bits while walking the heap. The heap type bits corresponding to an object is set by the allocator when it allocates that object.

Linux exposes pmem to applications as a file. Byte-level load/store access to pmem is available once a pmem file has been mmap'd to an application's address space [33]. We do all this in Go runtime and abstract these out through a Go package called *pmem*. We have incorporated a simple approach to enabling pmem support in Go - one with minimal changes to the design of the existing memory allocator and garbage collector (GC). Rather than make all the runtime data-structures crash-consistent, we log additional metadata to keep track of allocated regions. This saves us from extensive code changes that would have been necessary in the runtime and aids in the implementation of a robust design. A novelty of our approach is the minimal amount of additional metadata that we log.

4.3.1 Growable Heap Design

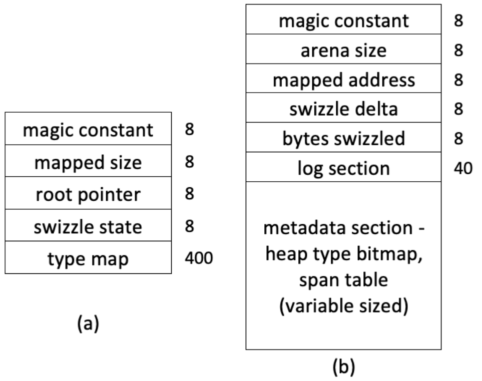


Figure 3: (a) pmem file header (b) arena header layout. Storage space in bytes adjacent to each field.

The runtime maps the pmem file into memory in arenas of sizes that are a multiple of 64MB. Figure 3(a) shows the layout of the global header stored in the beginning of the pmem file. *magic constant* is an 8-byte random number which helps to distinguish between first and subsequent initialization of persistent memory. *mapped size* is the size of the file currently mapped. *root pointer* is used to store the pointer to the application named objects (§4.4.1). *swizzle state* is used for implementing pointer swizzling (§4.3.2). *type map* is used to cache a fixed number of data types which are the most frequently allocated, helping pmem allocations of such types to be completed significantly faster (§5.1.5).

Each arena is divided into two sections - the arena header section and the region managed by the allocator. Figure 3(b) shows the layout of the header section in each arena. *arena size* is the size of the arena and *mapped address* stores the address at which the arena is currently mapped. *swizzle delta* and *bytes swizzled* help to implement the pointer swizzling algorithm. The *log section* helps to implement a minimal undo log in runtime which is used only during swizzling. The *metadata section* stores the runtime metadata for this arena as mentioned in §5.1.3. Whenever runtime runs out of persistent memory space, the persistent memory file is grown to accommodate a new arena. The metadata in the global header is updated in a consistent manner to reflect the addition of this new arena.

4.3.2 Pointer Swizzling

Pointer swizzling is a powerful feature that allows direct pointers to be stored in persistent memory and dereferenced, even after multiple invocations of the application. Since persistent memory is exposed through files mapped into memory, the virtual address of the mapping can change during each invocation. In the common scenario, go-pmem is able to map arenas at the same address, avoiding the need to swizzle pointers. If the mapping address changes, then all pointers stored in the pmem heap becomes garbage. Pointer swizzling is the process of 'fixing' these pointers by re-writing them with their new mapped address. Some libraries such as PMDK work around this problem by using a *base, offset* pair object as a reference to an object in persistent memory. Dereferencing such an object involves a hashmap lookup and offset computation which can get expensive.

Swizzling is done during pmem initialization in a per-arena manner. The algorithm uses the *swizzle delta* field in arena header to store the offset by which pointers that point into this arena should be changed. *bytes swizzled* help track how many pointers have already been swizzled. The *log section* in the arena help update pointers transactionally. The swizzling algorithm is resilient to crashes during swizzling. If a crash occurs, any partially executed swizzling is completed on the next run, before swizzling all pointers to the new mapped address. go-pmem uses a parallelized algorithm to swizzle pmem arenas. As an added advantage, the swizzle algorithm also implements pointer safety. On application restart, any pointer stored in the pmem heap that point outside the pmem heap are garbage. The swizzle algorithm zeroes out any such pointers to ensure applications do not access such rogue pointers. This gives users the freedom to store both pmem and volatile pointers in the pmem heap.

4.4 Restarting After a Crash/Exit

Our design currently handles graceful exits and non-corrupting failures. Listing 2 shows code starting/recovering from a crash. We provide a *pmem* package that handles initializing pmem and restarts.

```
package main
import "pmem"
func init() {
  firstInit := pmem.Init("database")
  var head *node
  if firstInit {
    // Create a named object called root
    head = (*node)(pmem.New("root", head))
  } else {
    // Retrieve the named object "root".
    // One-line restart!
    head = (*node)(pmem.Get("root"), head)
  }
}
```

Listing 2: Code for start/restart of application

4.4.1 Roots/Named Objects: pmem Package

Any data in volatile memory is lost on restart, so volatile pointers pointing to data in pmem will be lost too. This means only pointers residing in pmem pointing to data in pmem can be used to access pmem-resident data after a restart. We allow the applications to retrieve these pointers through string names. These can then be used to navigate other objects stored in pmem. We call these objects *"named objects"* and they can be pointers to native types, structs, or Go slices. Any updates to these named objects must be made through *pmem* package APIs as shown in Listing 2.

4.5 Transactions as a Part of Go: txn Block

To make sure no data is left in an inconsistent state, we use transactions. We change Go compiler to natively support undo transactions. We pursue transactions in Go with the intention of providing crash consistency and durability for updates to data in pmem. We introduce a new keyword to Go called *txn* that automatically intercepts stores to pmem and logs them in an undo log. We add a new SSA (Static Single Assignment) pass to Go compiler's backend that injects statements to Go's intermediate representation of the user code. Our design is derived from a similar technique used in Go compiler to add write barriers for garbage collection to stores in volatile heap [14]. Our new *LogStore* SSA pass comes after most of the existing Go SSA passes, so we don't lose on the existing optimizations. For example, successive stores to a pmem resident location can be eliminated by Go's *deadstore elimination* SSA pass even after our changes.

To demarcate transactions, we require users to contain their code within a *txn("undo")* code block. The *"undo"* indicates we currently support automatic code generation only for undo logging. We briefly discuss how we provide typical transaction properties below.

1. Atomicity: The *pmem.Init()* call (see Listing 2) initializes pmem and reverts any incomplete updates stored in the transaction logs of an application in case of a restart.
2. Consistency: We rely on the user to explicitly demarcate updates to pmem-resident data by using a txn block around the code.
3. Isolation: Simultaneous transactions accessing common data must not see any updates till a transaction is committed. We do not support isolation through software transactional memory but rely on programmers to use Go's mutex locks for critical sections. Our concurrency model is simple:
   (a) All the locks acquired within a transaction must be released within the transaction.
   (b) All the locks acquired outside a transaction must be released outside the transaction.

   Our design then makes sure that all the updates to shared data structures are visible only at the end of a transaction. This is achieved by delaying unlocking any locks acquired within a transaction until the end of the transaction. This is similar to the 2PL locking strategy used in database transactions [40].
4. Durability: All the changes made to data in pmem are made durable at the end of a transaction by flushing relevant data stored in the processor caches or buffers.

```
 1  // txn block of addNode()
 2    tx.Begin()        // <-
 3    mutex.Lock()
 4    tx.Log(&n.prev)  // <-
 5    n.prev = tail
 6    updateTail(tail, n)
 7    tx.End()          // <-
 8    mutex.Unlock()   // <- generated after End()
 9
10  // txn block of updateTail()
11    tx.Begin()        // <-
12    if inPmem(&tail.next) { // <- extra check
13      tx.Log(&tail.next)     // <-
14    }                        // <-
15    tail.next = n
16    tx.End()          // <-
```

Listing 3: Compiler generated code for listing 1.

We point out a couple of limitations that our locking model introduces -

1. Multiple *lock()* and *unlock()* operations on the same lock do not work inside a *txn* block.
2. Holding all locks taken inside a transaction until the end of the transaction can make lock-based critical sections in go-pmem slower than other models such as Atlas [27] that allow dependent transactions to run concurrently (as soon as required locks get unlocked). In order to provide isolation, they capture transaction dependency between multiple threads in their logs.

Listing 3 shows the compiler generated code of the `txn{}` code block in listing 1. Additional code added by the compiler to ensure transactional semantics is highlighted. Line 8 of listing 3 shows how a mutex unlock is delayed until the end of the transaction. go-pmem also allows function calls within a transaction, as explained further in §5.2.2.

4.6 No Persistent Data Types

go-pmem intentionally does not introduce new data types. Previous works have often introduced data types like *pint*, *p<int>* or *persistent int*. They usually do this because they offer a library implementation and overload the assignment operator [18] or they want type-safety, reference counting etc. for pmem resident data [32]. We believe this complicates the programming model and instead rely on Go's in-built typesafety and garbage collection.

5 Implementation

go-pmem adds about 4000 lines of code and removes 300 lines of code from Go runtime, excluding code documentation. This does not include the code in the CPUID package from Intel [10] that is used by runtime to identify CPU features for flushing CPU caches. The two Go packages (pmem and transaction) took close to 2300 lines of Go code. These changes were made on top of Go 1.11 release and don't include the code for testing these changes. Our implementation currently works only for 64-bit Linux 4.15 and above.

5.1 Runtime Details

5.1.1 Data Structure Support

To support persistent memory allocations, runtime datastructures such as the mspan, mcache, mheap, mcentral were extended to store persistent memory metadata. The mspan structure was augmented to identify if this is a pmem span or volatile memory span. Similarly, mcache, mcentral, and mheap were doubled in size to store pmem spans separately. It should be noted that no Go runtime data-structures are stored in pmem. Instead, minimal additional metadata is logged in pmem arena header to capture pmem state.

5.1.2 Memory Allocation

A persistent memory allocation request results in the following workflow - if a cached pmem span is available in the mcache with a free slot, it is used to satisfy the allocation request. Otherwise, a new pmem span is requested from the mcentral/mheap. If no pmem span is available, a new pmem arena is mapped to memory. The required span is then carved out from the pmem arena to satisfy the allocation request. Any required metadata is also logged (§5.1.3). An advantage of the go-pmem allocator is that these steps can be done without invoking transactions. This is because any pmem leaks are plugged by the GC during heap recovery (§5.1.4).

5.1.3 Metadata Logging

The metadata that is logged is the minimum amount of information that the runtime needs to reconstruct the memory allocator and garbage collector state on the subsequent execution of the application. The benefits of keeping the metadata to a minimum are twofold: we do not need to introduce complex transactions in the runtime to maintain consistency and the allocator performance is only slightly affected due to the additional logging.

Two kinds of runtime metadata are logged. The first is the GC heap type bits. Whenever the allocator sets heap type bits for a pmem object, it is also logged in the pmem arena header section. Like its volatile memory representation, heap type bits occupy 2 bits for every 8 bytes of heap data. In §5.1.5 we talk about an optimization that helps avoid heap type bitmap logging for frequently allocated data-types. The second is the span table. The span table captures information such as which spans are in use and the size class of the span. Span table logging happens when a new span is created by the allocator or freed up by the GC. Span table reserves 32 bits for every pmem page in the arena (Go uses a page size of 8KB). For a 64MB pmem arena, the arena header section occupies 2024KB, and 63512KB is available for the allocator. The arena section includes 80 bytes for the header, 31756 bytes for the span table and 1984.75KB for the heap bitmap, making the pmem arena memory overhead as 3.09%.

5.1.4 Reconstruction

All runtime data structures related to memory allocator and garbage collector are stored in volatile memory. So, if an application crashes, all state information is lost. *Reconstruction* is the term used in our project to denote the process of bringing back the state of the memory allocator and garbage collector related to persistent memory as it was before the crash. The additional metadata stored in pmem aids in this *reconstruction* process. No persistent memory data is modified during reconstruction. Hence the reconstruction process is resilient to any crashes that happen while it is ongoing. Briefly, *reconstruction* works as following:

1. GC is disabled until *reconstruction* finishes so that it does not interfere with the *reconstruction* process.
2. Spans are recreated using the logged span metadata table.
3. The logged heap type bitmap is copied as-is to the arena metadata in volatile memory.
4. Pointers are swizzled, if necessary
5. GC is re-enabled

GC walks the *reconstructed* pmem heap starting from the named root objects. It marks all reachable objects as being in-use, and makes any leaked pmem objects available for reuse. This GC walk runs in the background, making the *reconstruction* process execute quickly.

5.1.5 Using Go's Typesystem to Optimize Pmem Allocations

An allocation of an object that has pointers in it results in the allocator setting the heap type bits for it. If this is a pmem object, these type bits are also logged in the pmem arena header. If a datatype is heuristically found to be a frequently allocated type, then that type is promoted to be cached specially in mcache to speed up allocations of such objects. The heuristic used is the following - the number of allocations of such an object has exceeded the number of slots available in a span corresponding to this object sizeclass and its allocation frequency is greater than 100 objects per second. A span specially so cached is used only to allocate objects of one type. This makes it possible that the heap type bits be logged only for the first object allocated, making further allocations from this span very fast. We employ the typemap (§4.3.1) in the global header to store what types have been specially promoted. Our design supports promoting up to 50 types to be specially cached. In our experience, the maximum number of types frequently allocated by various pmem applications were far fewer. Each type is represented in the typemap section in the pmem header using an 8-byte identifier, making typemap occupy a total of 400 bytes.

5.1.6 Undo Log Implementation

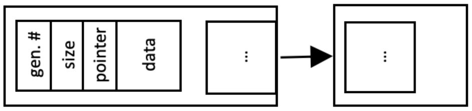


Figure 4: Undo log design

Undo logs are stored within a linked list of Go's byte arrays. Each undo log entry has the layout as shown in figure 4. *gen.#* stores current undo log generation number. On a successful abort/commit, the *generation number* is bumped up to mark all entries as no longer valid. *size* stores the size of the data logged. *pointer* is the address at which this data originally resided. *data* contains the logged copy of the data. Since we do not anticipate a transaction abort in the common case, log entries are populated using *movnt* instructions so that data is directly moved to the pmem device bypassing the processor cache.

We also employ a number of optimizations to make logging fast in go-pmem:

1. Empty transactions do not incur any runtime overhead as no cache flushes or memory fences are issued.
2. Logging the same object multiple times incur minimal overhead. We maintain a map to track what objects have already been logged.
3. As byte arrays are very common in Go, we specially optimize logging 1-byte data. We try to pack consecutive 1-byte data objects in a single log entry rather than create separate entries for each.

5.1.7 Working Around Go's GC to Optimize Undo Logging

Go uses a mark-and-sweep GC. It scans the heap in a breadth-first fashion, traversing the heap through the live pointers it finds. The scan starts from the pointers found in goroutine stacks and global variables. We want objects pointed at from the logged data to be kept alive until the transaction completes. But since data is logged in a byte-array, runtime no longer has the type information of each data item logged. Our initial logging library used Go's reflect package which gave a poor performance, so we decided to use Go's byte arrays. To identify pointers in the byte array, whenever a data is logged, all pointers within this data item is stored separately in an array of pointers residing in volatile memory. This ensures that GC finds these pointers while traversing the heap.

5.1.8 Cache Flushing

Data written to persistent memory can be guaranteed to be persistent only after they are flushed from the processor cache to the persistent memory media. The runtime provides the PersistRange API to flush the processor cache over the address range passed to it.

```
func PersistRange(addr unsafe.Pointer, len int)
```

If the persistent memory device supports direct-access, this function takes care of executing the most optimized cache flush instruction supported on the processor (such as clwb, clflushopt, or clflush) and any necessary memory barriers [42]. If the device does not support direct-access, then *PersistRange* invokes the msync system call to flush data at a page size granularity. Transactions in go-pmem automatically call into the runtime to flush data from the caches, freeing the programmer from having to do it manually.

5.2 LogStore SSA Pass

The LogStore SSA pass can automatically interpose stores to persistent memory and redirect this to an undo transaction. The user can wrap any codeblock with a *txn("undo")* keyword and engage this new SSA pass. In the absence of any txn block, this SSA pass does not do anything. Because we wanted to keep the changes to a minimum, this SSA pass can be plugged into/out of the Go compiler's usual workflow.

5.2.1 Handling Volatile Memory Access Inside Transactions

Go uses escape analysis [28] to figure out life time of variables, and thereby avoids unnecessary memory allocations to volatile heap. We extend this to avoid unnecessary allocations to persistent heap and track pointers in volatile heap. With this static analysis, we know the probable location of an update within the transaction. If this update can be proven to be a location in volatile memory, we do not do anything. Otherwise we store the current value of the data in a persistent log which will be replayed in case of a crash.

5.2.2 Handling Function Calls Inside Transactions

Go compiles each function independently and cannot know if this method will be called from a transaction, or a non-transactional code at compile time. Instead of cloning the function for transactional access, we maintain a per Go-routine handle and ask the user to wrap any potentially transactional code within a txn code block. Based on whether a transaction is already ongoing or not, we intelligently start a new transaction or continue the same transaction at runtime. In case this function is called from a non-transactional code, the function simply operates on volatile data without any side effects. Using the techniques mentioned in §5.2.1 we try to keep the performance overhead to minimum in this case.

5.3 Implementing Go-redis-pmem

We implement a multithreaded feature-poor redis server called *go-redis-pmem* written using go-pmem. *go-redis-pmem* currently supports storing/retrieving string KV pairs in pmem and can run traffic generated from memtier benchmark. It currently has 6800 lines of code across 11 files and 360 functions. As we were implementing go-redis-pmem, we realized that as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory. Our desire to support function calls and ability to reuse functions for data in volatile and persistent memory was driven by the implementation of go-redis-pmem and the ease this offered to the programmer.

5.4 Limitations

go-pmem is a work in progress and will continue to evolve as we gain more experience in programming applications for persistent memory. Below, we enumerate the current limitations of go-pmem:

1. No support for shrinking persistent heap file after they have grown. We believe an offline application working like a compacting garbage collector can fix this.
2. No support for allocating/transactionally using Go's maps or channels in pmem. go-pmem currently supports basic Go types, structs, and Go slices.
3. No support for working with multiple persistent heaps
4. No support for operating systems other than 64-bit Linux.
5. No support for redo/custom implementation of transactions with the txn code block.
6. We do not handle the case when there are traditional I/O operations (like network, display, etc.) inside a transaction and then there is a crash. Previous works have handled this by throwing an exception [34] and by providing safe IO [44]. We can do something similar.

6 Evaluation

We use the benchmarks from Computer Language Benchmarks Game (CLBG) [5] as the microbenchmarks to compare the performance of our memory allocator and transactions.

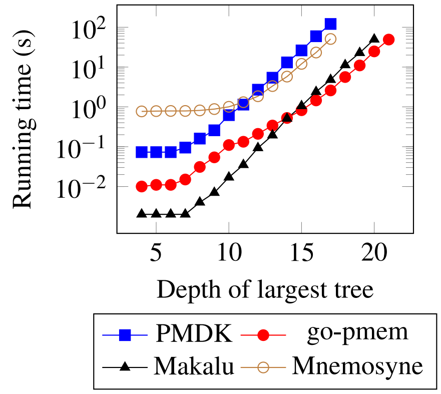


Figure 5: Runtime as depth of largest tree changes

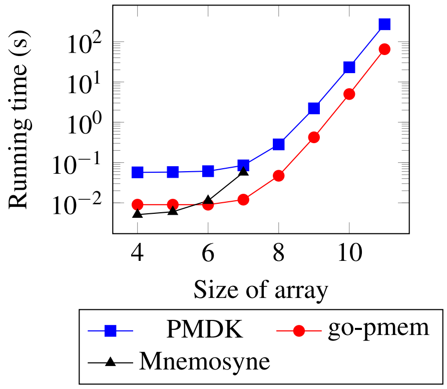


Figure 6: Runtime as num. of permutations in fannkuch-redux changes

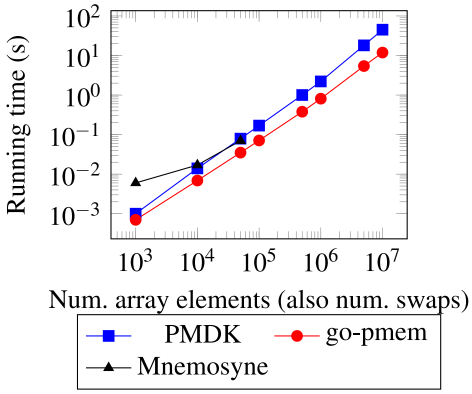


Figure 7: SPS Benchmark: Overhead of transactions

CLBG has gained wide attention for comparing the performance of different programming languages [39]. Go has also used some of these benchmarks to optimize their implementation in the past [23]. We extend some of these benchmarks to test how they perform when run on pmem, and don't focus on others as they test other language features such as arithmetic precision, hashtable performance etc. which is not our focus here. To test multicore scalability of go-pmem, we use the microbenchmarks from the Phoenix suite [41]. The phoenix suite was originally written to evaluate the MapReduce model for multi-core systems. We use the *pthread* version of these benchmarks to port to various pmem libraries. We also compare the performance of *go-redis-pmem* with other Redis implementations. In these evaluations, we ensure there is no remote persistent memory traffic by keeping only one CPU socket on. We compare our work against PMDK stable version (1.7 release commit *bc5e30948*) and we write code in C++ using C++ bindings from PMDK (stable version 1.8 commit *ab4ff69b7*) [22]. The Mnemosyne examples build on implementation code from [7] and Makalu examples run on implementation code from [16].

6.1 Experimental Setup

Our system is a 24-core Intel Cascade Lake machine with hyperthreading disabled and only one socket to avoid remote pmem traffic. It has 4 Intel© Optane™ DC Persistent Memory Module, each of 128GB, and 64GB of DRAM. In all the runs, the deviation observed across runs was <2%. We report the average runtime across 3 runs.

6.2 Change in Compile Time

The compilation time of Go source code increased by 3.4% (from 42.71s to 44.16s) because of all our changes to Go compiler. In the compilation of go-redis-pmem (which has 6800 lines of code and 11 files), we did not see any noticeable difference in the compile time with and without the new ssa pass. The difference was <1% (0.71s vs 0.713s). These numbers were obtained for a fresh compilation with go's build cache cleaned. With the build cache enabled, the observed difference was even smaller (less than 1ms).

6.3 Memory Allocator Performance

The Binary Tree allocator microbenchmark from CLBG stresses the pmem allocator by creating several perfect binary trees. One of these stresses memory to see maximum memory available. One is a long-lived binary tree and there are several short-lived trees which are created and then deallocated. Figure 5 compares the performance of go-pmem's pmem allocator to PMDK, Mnemosyne and Makalu. We note that because go-pmem can garbage collect pmem we did not have to free any tree nodes. Makalu and go-pmem do not write to pmem for each new allocation and so are at least an order of magnitude faster than PMDK and Mnemosyne. PMDK must do all allocations and deallocations within a transaction and performs the worst.

6.4 Performance of Transactions

6.4.1 Long Running Transactions

We use *fannkuch* from CLBG and *sps* to model programs with a long-running transaction. *Fannkuch* takes a byte array of size *n* and shuffles around elements for all *n!* permutations of this array. We model all this to happen inside a single transaction. This lends it the behavior of a long running transaction. We also noticed that both PMDK and go-pmem use undo logs and maintain minimal state in pmem (only the oldest value of the array) whereas Mnemosyne uses redo log and quickly starts to use a lot of pmem. The *sps* microbenchmark has been used previously [30, 31] to report throughput of transactions. *sps* randomly performs swaps between entries of an integer array. The number of these swaps is equal to the number of elements. We change the number of elements in this array from 1k to 10 million and do all the swaps within one single transaction. So unlike *Fannkuch*, the size of data being operated on also increases as the transaction becomes longer. Figures 6 and 7 compare the running time of *Fannkuch*

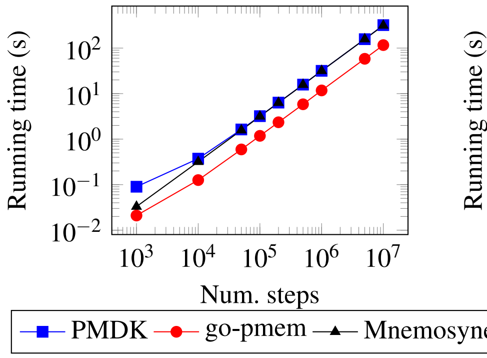


Figure 8: n-body: Runtime as num. of steps a planet moves varies

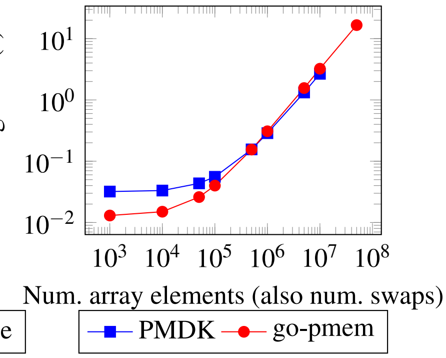


Figure 9: Tx recovery time after abort at the last swap

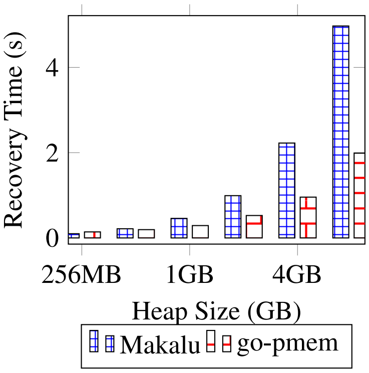


Figure 10: Restart time comparison using fill-heap

and *sps* for PMDK, Mnemosyne and go-pmem. We note that Mnemosyne curves stop early as the public implementation does not support large amounts of data within a transaction. Similar results for Mnemosyne were reported in [31]. The PMDK curves stop early as the default PMDK implementation does not allow creating a pmem file of 2GB or larger. For both these cases, we consistently outperform PMDK by 3-4x as the transactions become larger.

6.4.2 How Much Data do Applications Store to pmem?

We also want to highlight that the existing programming models are inept for long-running applications like the ones we model above. PMDK asks the user to specify the pmem file size at the beginning. It crashes if this size is exceeded. Mnemosyne always creates a pmem file of fixed size. We think it is very difficult to predetermine the pmem capacity that commercial applications will use. With the ability to grow pmem heaps (§4.3.1), our programming model is more flexible.

6.4.3 Several Short Transactions

We use *n-body* microbenchmark from CLBG to model a program with several short transactions. For short transactions, the overhead in setting up the transactions is not amortized, and we try to capture this overhead here. n-body models the orbits of planets using an algorithm. The input is the number of steps that the planets move from a starting point and the output is the new coordinates of the planets. We change one step movement of the planets to be one transaction. Figure 8 shows the running time as we vary the number of small transactions. go-pmem consistently performs 2-3x faster than PMDK and Mnemosyne even as the number of small transactions in the application increase to 10 million.

6.5 Multicore Scalability

We run all 7 benchmarks from the Phoenix 2.0 suite [41] to evaluate how go-pmem scales on multithreaded benchmarks. We modified these benchmarks to keep the data manipulated by each thread in pmem. Figure 13 captures the relative running time of PMDK compared to go-pmem. All benchmarks use 24 threads and run on the largest input configuration provided by Phoenix. Unlike PMDK and go-pmem, Mnemosyne stores more data in its redo logs and we were not able to get it running on any of these benchmarks. go-pmem scales much better than PMDK on all benchmarks other than *linear regression*. *Linear regression* stores very little data on pmem and incurs minimal transactional overheads. The benchmark *kmeans* uses 2D arrays in pmem and is significantly slower for PMDK. PMDK uses fat pointers and accessing 2D arrays requires multiple indirections. Although figure 13 shows results with 24 threads, we ran the benchmarks varying the number of threads from 2 to 24. The results with different thread counts are similar to the results shown in figure 13.

6.6 Restart Time Comparison

6.6.1 Undo Transaction Recovery Time After a Crash

We reuse *sps* benchmark from §6.4.1 to measure time spent by undo transactions in PMDK and go-pmem to revert back to consistent application state. We change the number of integers swapped in *sps* and crash at the last integer swap. Figure 9 shows go-pmem performs at par with PMDK when the amount of data to recover is less but gets slower by 20% as the amount of data to recover increases. We have not optimized the recovery path too much as this is not the common path. One obvious optimization is to store application data in cacheline aligned chunks. This will reduce the number of writes to pmem when we write back the consistent data during restart. Optimizing this path without affecting the performance of common cases remains on our future agenda.

6.6.2 Restart Time of Persistent Heaps

We use *fill-heap* benchmark from Makalu [26] to compare the restart time as the size of the persistent heap changes. *fill-heap* creates 64-byte objects to fill up a specified heap size. It makes half of these objects reachable from the pmem root objects. On restarting the *fill-heap* application, we measure the time taken by go-pmem, PMDK, and Makalu to recover

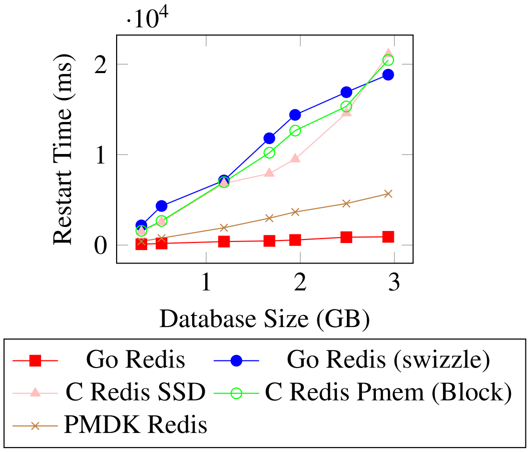


Figure 11: Comparing restart time of various Redis versions

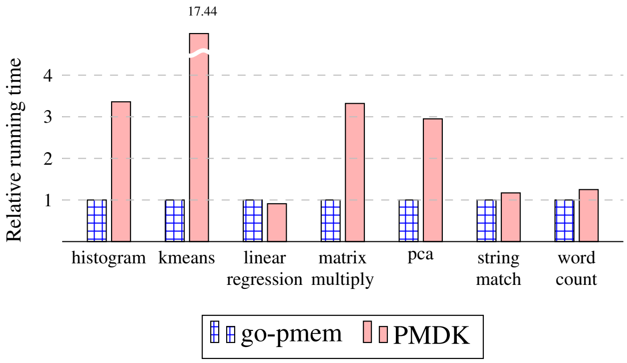


Figure 13: Relative comparison on Phoenix benchmarks

the pmem heap. As seen in figure 10, go-pmem recovers the pmem heap much faster than Makalu as Makalu has to go through an expensive offline GC phase. PMDK recovers almost instantaneously, because their allocator is inherently transactional, and hence incurs minimal startup cost.

6.6.3 Restart Time of Redis Variations

To measure how go-pmem heap's recovery fares on a pmem application, we compared the restart time of go-redis-pmem against various other Redis configurations as shown in figure 11. Go Redis swizzle measures the cost of swizzling pointers by force mapping all arenas at a different address than where it was originally mapped. C Redis SSD persists its data as an AOF file on SSD, whereas C Redis Pmem block persists the AOF file on pmem used in block IO mode.

6.7 Go Benchmarks

We run a set of 4 macro-benchmarks used by Go community to monitor Go performance regressions as new features are added to the compiler [9]. These benchmarks stress the memory allocator, GC, compiler, etc. Table 2 compares performance of go-pmem versus Go-1.11 upon which our changes are based on. Our changes add little to no performance difference in these benchmarks.

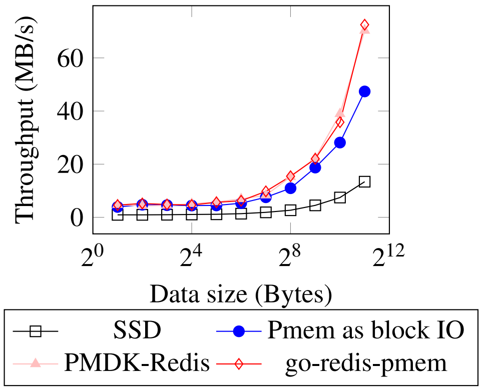


Figure 12: Redis throughput comparison against memtier benchmark

Benchmark	go-1.11	go-pmem	delta
Build-24	23.9s	23.9s	0%
Garbage-64	23.4ms	23.6ms	0.87%
JSON-24	84.0ms	84.3ms	0.44%
HTTP-24	73.9μs	75.5μs	2.15%

Table 2: Go benchmark comparison

6.8 Go-redis-pmem

Figure 12 shows the throughput of go-redis-pmem on the same memtier benchmark used in figure 1. Even though go-redis-pmem is multithreaded this configuration uses one client thread for a fair comparison. We can see that go-redis-pmem matches the performance of pmdk-redis reconfirming our observation that if applications use persistent memory in byte addressable mode, they will perform the best. In data not shown here, we did see that go-redis-pmem performed better than pmdk-redis and redis-3.2 when there are multiple client threads because it is multithreaded.

7 Conclusion

In this work, we presented go-pmem, an opensource extension to the Go language that allows programmers to develop pmem applications in Go. go-pmem is a natural extension of Go for pmem support, following the normal idioms of the Go language. We present a simple programming model and demonstrate its effectiveness by developing a feature-poor Redis equivalent key-value store in Go.

Acknowledgements

We thank the anonymous reviewers and our shepherd Changhee Jung for their invaluable feedback.

References

[1] Available first on google cloud: Intel optane dc persistent memory, google cloud blog. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[2] A black hole file system that behaves like /dev/null. https://github.com/abbbi/nullfsvfs.

[3] A brief retrospective on transactional memory. http://joeduffyblog.com/2010/01/03/a-brief-retrospective-on-transactional-memory/.

[4] A call for a data persistence study group. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1026r0.pdf.

[5] The computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[6] Direct access for files, kernel.org. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[7] Gcc port of mnemosyne. https://github.com/snalli/mnemosyne-gcc/tree/cfed43142cdcb5175f1b7c75cd6a922ce561060e.

[8] The go prgramming language, google. https://golang.org.

[9] Golang benchmarks. https://github.com/golang/benchmarks/.

[10] Intel corporation. cpuid library for go prgramming language. https://github.com/intel-go/cpuid.

[11] Intel corporation. pmem-redis. https://github.com/pmem/redis.

[12] Intel optane$^{\text{TM}}$ technology, intel. https://www.intel.com/optane.

[13] Intel vtune profiler, intel. https://software.intel.com/en-us/vtune.

[14] Introduction to the go compiler's ssa backend. https://github.com/golang/go/tree/master/src/cmd/compile/internal/ssa.

[15] Jep 352: Non-volatile mapped byte buffers, openjdk. https://openjdk.java.net/jeps/352.

[16] Makalu : Nvram memory allocator. https://github.com/HewlettPackard/Atlas/tree/makalu/makalu_alloc.

[17] Persistent collections for java. https://github.com/pmem/pcj.

[18] Persistent memory development kit. https://pmem.io/pmdk/.

[19] Persistent memory documentation. https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk.

[20] Persistent memory in linux, snia. https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Coughlan_Tom_PM_in_Linux.pdf.

[21] Persistent memory wiki. https://nvdimm.wiki.kernel.org/.

[22] Pmdk c++ bindings. https://github.com/pmem/libpmemobj-cpp.

[23] Programs from "the computer language benchmarks game", used to be in the main go distribution in test/bench/shootout. https://github.com/golang/exp/tree/master/shootout.

[24] Redis labs. https://redis.io.

[25] Redis labs. redis and memcached traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[26] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016.

[27] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[28] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10):1–19, 1999.

[29] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 197–212. ACM, 2013.

[30] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 47(4):105–118, 2012.

[31] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.

[32] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. Nvl-c: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 125–136. ACM, 2016.

[33] SNIA NVM Programming Technical Working Group et al. Nvm programming model (version 1.2), 2017.

[34] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Sigplan Notices*, volume 38, pages 388–402. ACM, 2003.

[35] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, 2017.

[36] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[37] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[38] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[39] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In *ACM SIGPLAN Notices*, volume 52, pages 120–131. ACM, 2016.

[40] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

[41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[42] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.

[43] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–332. ACM, 2019.

[44] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M Swift, and Adam Welc. xcalls: safe i/o in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 247–260. ACM, 2009.

[45] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[46] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 70–83. ACM, 2018.

End the Senseless Killing: Improving Memory Management for Mobile Operating Systems

Niel Lebeck
University of Washington

Arvind Krishnamurthy
University of Washington

Henry M. Levy
University of Washington

Irene Zhang
Microsoft Research

Abstract

To ensure low-latency memory allocation, mobile operating systems kill applications instead of swapping memory to disk. This design choice shifts the burden of managing over-utilized memory to application programmers, requiring them to constantly checkpoint their application state to disk. This paper presents Marvin, a new memory manager for mobile platforms that efficiently supports swapping while meeting the strict performance requirements of mobile apps. Marvin's swap-enabled language runtime is co-designed with OS-level memory management to avoid common pitfalls of traditional swap mechanisms. Its key features are: (1) a new swap mechanism, called *ahead-of-time (AOT) swap*, which pre-writes memory to disk, then harvests it quickly when needed, (2) a modified bookmarking garbage collector that avoids swapping in unused memory, and (3) an object-granularity working set estimator. Our experiments show that Marvin can run more than 2x as many concurrent apps as Android, and that Marvin can reclaim memory over 60x faster than Android with a Linux swap file can allocate memory under memory pressure.

1 Introduction

Over the past decade, mobile apps have become bigger and more complex [28], far outpacing increases in mobile device memory [4]. This trend has increased memory pressure on mobile operating systems as apps compete for limited space. Going forward, mobile OSes must more efficiently share memory across demanding apps, or user experience will suffer.

Unfortunately, while mobile apps have become more sophisticated, mobile memory management remains in its infancy. Today's popular mobile OSes set a fixed upper bound on memory for each running application (e.g., 1.4GB for iOS running on an iPhone X [36] and 512MB for Android running on a Google Pixel XL). They never overcommit memory; instead, they kill running applications and restart them later.

This simplistic approach worked well when mobile apps were small and largely stateless. However, it is unsustainable as mobile OSes replace desktop ones (e.g., Android is now the most used OS in the world) and mobile apps replace desktop counterparts (e.g., Google Docs and Word Online replacing Microsoft Word). Today's apps already do not fit into their memory allocation, so they manually swap objects between memory and local storage or use libraries to meet their needs [11]. Because apps are increasingly likely to be killed due to memory pressure, they must also continuously save execution state to disk and strive to minimize their start-up times to cope with frequent restarts. Despite significant engineering effort [21, 27, 32], it still takes several seconds to kill and restart popular apps.

Improving mobile memory management is difficult. Mobile apps run in high-level language runtimes (e.g., Swift, ART), which limit OS insight (e.g., working set estimation is impossible) and are notoriously difficult for OS-level memory managers to work with [20]. Further, mobile apps often allocate large amounts of memory quickly (e.g., when starting, or for cloud downloads); unless the OS keeps a large pool of free memory, this is easier to accommodate by killing entire applications. Finally, touch-based interfaces impose strict latency requirements, which swapping to disk cannot meet.

In this paper, we improve mobile memory management with a key observation: unlike other operating systems, mobile OSes run all of their apps in a *common* language runtime. For example, all apps running on Android must run in the Android Runtime (ART). This difference lets us co-design the language runtime to assist the mobile OS in optimizing memory management instead of hindering it. Due to its knowledge of memory usage, the language runtime becomes an ideal place for mechanisms that can better manage memory.

This paper demonstrates the value of leveraging the runtime for OS tasks. We present *Marvin*, a new memory manager for Android that efficiently supports memory overcommit. Marvin implements most memory management in the language runtime, which has more insight into an application's memory usage. Marvin relies on the operating system only for cross-application resource allocation.

By integrating with the language runtime, Marvin can offer

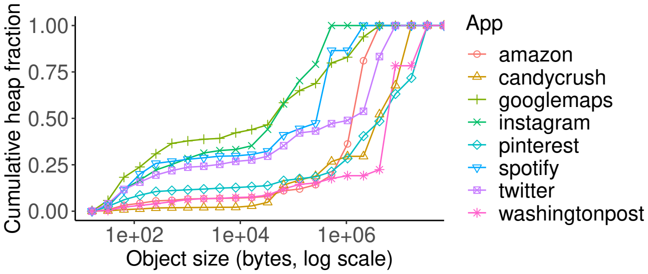


Figure 1: CDF of object size and heap percentage occupied by objects that size or smaller.

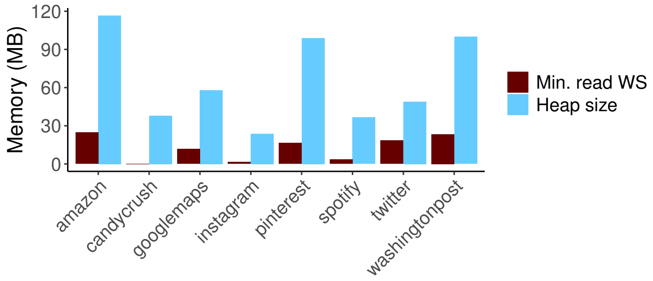


Figure 2: *The cost of fixed allocation.* Each bar shows the total Java heap size of a popular app alongside its minimum Java working set during active use.

three new features that enhance memory management:

- A new swap mechanism, which performs *ahead-of-time* swapping to disk to avoid synchronous disk writes when reclaiming memory, and which leaves checkpointed objects in memory (unlike Linux's kswapd [18]).
- A new object-level working set estimator, which separates garbage collector (GC) and app accesses, and avoids false sharing with object-level access tracking.
- A new bookmarking garbage collector [20], which tracks exact liveness data without accessing swapped-out objects.

We implement a prototype of Marvin by modifying the interpreter and compiler of the Android Runtime (ART). Experiments show that our Marvin prototype is able to run more than 2x as many concurrent apps as Android, and that Marvin can reclaim memory over 60x faster than Android with a Linux swap file can allocate memory under memory pressure.

2 Limitations of Modern Mobile OS Memory Resource Management

Although mobile OSes may be based on traditional OSes (e.g., Android and Linux), they diverge in two important ways: (1) for each app, they bound memory usage to a fraction of physical memory (e.g., 512MB on a 4GB device), rather than letting apps allocate as much memory as they need, and (2) they kill applications when physical memory runs out rather than overcommitting memory through paging or other mechanisms. To motivate our work, we ran experiments with popular apps that show the reasoning and cost for these design decisions. We ran all experiments on a Pixel XL phone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU.

2.1 Fixed Memory Allocation

Mobile OSes have poor insight into app memory usage. The runtime garbage collector regularly touches all objects and moves objects for heap compaction, and the OS cannot distinguish this activity from app accesses. Mobile apps also access language-level objects, which vary in size, while the OS can only track memory accesses at page granularity. To understand the impact of object-level accesses on page-sized access tracking, we measured the size of objects in popular apps. Figure 1 shows a CDF of the size distribution. Most objects are not page-sized (e.g., up to 40% of objects are smaller than 4KB), so the OS cannot accurately track their usage.

Without good insight into app memory usage, today's mobile OSes allocate all apps a fixed memory budget. On Android, this memory limit is the same whether an app is in the foreground or background. Android attempts to minimize the memory footprint of apps using techniques such as forking all apps from a single "zygote" process with copy-on-write pages. These techniques reduce duplication of framework data structures and shared libraries but do not impact application objects in the Java heap. Using Marvin's object-level working set estimator, we measured the working set of popular apps. Figure 2 shows that although the heap footprint of these apps is large, their working sets actually account for a small fraction of their total heap size. This rarely accessed memory would be better utilized keeping other apps alive, rather than wasting space not being used.

Popular apps often have large memory footprints but small working set sizes because they cache as much as possible from the cloud. This caching improves performance, but it leads to poor memory utilization, and choosing the correct cache parameters is difficult [30]. Worse, modern applications frequently exceed their memory budgets. Coping with this problem requires apps to implement manual swap-to-storage, which adds significant programming complexity [10, 11]. While caching libraries like Glide [5] and Fresco [34] are helpful, they do not apply to all memory objects. Therefore, today's apps use a complex combination of libraries and manually shuffling data between memory and disk.

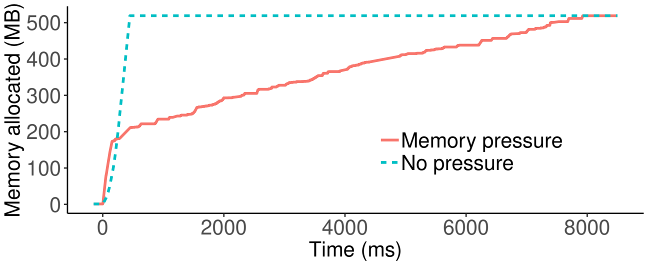


Figure 3: Progress over time of a memory allocation on Android with a swap file and memory pressure vs. no memory pressure.

2.2 No Memory Overcommit

Today's mobile OSes kill applications rather than swapping to disk when physical memory runs out. They take this approach because mobile apps must respond to user input within hundreds of milliseconds, so traditional swap mechanisms, which place synchronous disk writes on the critical path, impose too much latency. To measure the effect of swapping on memory allocation, we enabled a Linux swap file on our Android test device [35], and we measured the amount of time required to allocate 512MB when the Android OS had free memory and when it had a swap file and memory pressure. Figure 3 shows the progress of the memory allocation over time. With memory available, the OS allocated all 512MB in 450ms; however, with a swap file and memory pressure, it took almost 8 seconds for the OS to allocate the same amount. Such high allocation latency would be unacceptable if an app were allocating memory in response to user input.

Unfortunately, killing and restarting apps comes at a cost. As shown in Figure 2, modern apps have large memory footprints, and a restarted app must fetch all of its cached data from the network or disk. We measured the amount of time needed to restart popular apps and compared it to that needed to fetch their entire checkpointed memory image from disk. As shown in Figure 4, restarting apps takes 4-27x longer than fetching the all of the app's memory from disk.

The ability to kill and restart apps at any time also imposes a programming burden on app developers. Modern OSes give apps a limited time budget to perform cleanup before being killed. This limit leads apps to constantly write state to storage; in fact, Android encourages it [9]. Such constant checkpointing in response to app lifecycle events adds programming complexity, a challenge described in prior work [14]. Not only do app developers have to manage the checkpointing process, they have to correctly use a variety of mechanisms to do so with good performance [12]. The programming effort required to prepare for unexpected app deaths is an additional cost that app developers must pay.

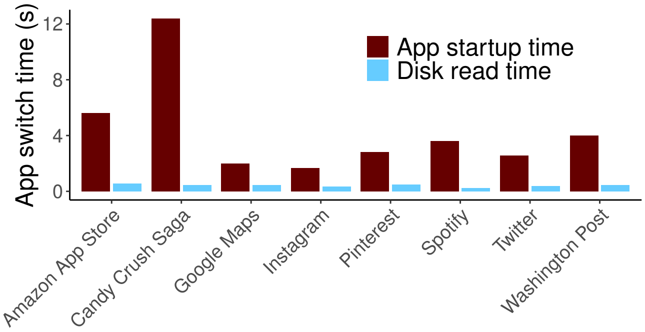


Figure 4: The cost of re-starting apps compared to reading their memory image from disk.

3 Our Approach

The primary barrier to improving memory resource management in mobile operating systems is the OS's lack of insight into the language runtime. To overcome this barrier, we *co-designed* the language runtime and the mobile OS. Mobile operating systems are uniquely suited to such co-design because, unlike their desktop counterparts (e.g., Linux), they force all applications to use the same language runtime.

Marvin's design focuses on Android (and the Android Runtime (ART)), which is now the most popular OS in the world [25]. Using the language runtime, Marvin manages memory entirely at object granularity, tracking, reclaiming and faulting in entire objects. This section describes in more detail the barriers to better memory resource management in a mobile OS and how Marvin addresses those challenges.

3.1 Object-Level Working Set Estimation

The first step to better memory resource management is a better understanding of each application's working set. Thus, Marvin implements language-aware working set estimation in the language runtime, which tracks app reads and writes at object granularity. Marvin uses this mechanism to identify candidates for ahead-of-time swap (Section 3.2) and separate garbage collector accesses from app accesses (Section 3.3). Lacking hardware access bits to help with this tracking, Marvin implements software access tracking in both the ART interpreter and compiler, as modern mobile language runtimes run both interpreted and compiled code.

3.2 Ahead-of-time Swap

As noted, swapping to disk when the OS needs memory is not feasible for mobile OSes and their touch-based apps. Marvin takes a different approach. While traditional swapping mechanisms write to disk when memory is needed, Marvin uses a new *ahead-of-time* swap technique. This technique saves memory to disk *before* it is needed and then reclaims those pages under memory pressure. Ahead-of-time swap separates swapping to disk from reclaiming memory; thus, we distinguish between *saved* objects, which have been copied to disk but still reside in memory, and *reclaimed* objects, which no longer reside in memory but are only on disk.

Swapping objects before they are needed leaves a large pool of clean memory that the OS can quickly reclaim and reallocate. While this technique lets apps continue using memory until the OS reclaims it, whenever the app dirties a page, the swap mechanism must update the on-disk copy before the OS can reclaim it. Due to this trade-off, Marvin prioritizes swapping objects that are infrequently or never written.

3.3 Bookmarking Garbage Collector

Like traditional swapping, ahead-of-time swapping is affected by friction with the language-level garbage collector. As noted by Hertz et al. [20], the garbage collector can inadvertently page in memory when walking the object heap to look for unused objects. With ahead-of-time swapping, the garbage collector can also inadvertently dirty pages when updating references, causing unnecessary writes to disk. Marvin solves this problem by integrating a modified bookmarking garbage collector [20] into the Android Runtime.

Marvin's swap mechanism leaves *stubs* – analogous to bookmarks – for each reclaimed object that detail the objects' references to other objects. Using these stubs, its garbage collector can process a reclaimed object during a mark-and-sweep run without faulting in the entire swapped object. Marvin's swap mechanism can further optimize swapping from disk by dropping dead objects without faulting them in.

4 Marvin Overview

Marvin is a new mobile memory manager that supports flexible memory allocation between apps and memory overcommit through swapping. Marvin includes components in the language runtime and OS, which are co-designed to provide better memory management. This section overviews both.

4.1 Design Goals

Marvin's design meets the following goals:

1. **Fast memory allocation.** Marvin must allocate memory quickly on-demand, avoiding disk accesses on the critical path for memory allocation.

2. **High memory utilization.** Marvin must provide the illusion of unlimited memory, provided working sets do not exceed the size of physical memory.

3. **Minimal overhead.** Marvin must impose low runtime overhead and require no app code changes.

While the last two goals are common to all memory management systems, existing mobile platforms sacrifice high memory utilization for fast memory allocation. Marvin aims to achieve all of these goals.

4.2 Marvin System Model

Marvin assumes a systems environment that meets three requirements: (1) all apps are written in a single managed language (e.g., Java), (2) all apps run in a single managed language runtime (e.g., ART), and (3) the runtime performs garbage collection or some form of automatic memory management. Marvin's design targets Android, which meets all of these requirements. Android also runs some libraries using native code; however, Marvin is not needed to manage their memory because they do not have the same issues with OS-level memory management as managed languages.

Marvin's optimizations could apply to other operating systems as well (e.g., iOS). For example, Swift uses automatic reference counting as an alternative to garbage collection, so it would require a bookmarking reference counter that can track references without faulting in the entire object.

Marvin runs unmodified Android apps on ARM64-based devices. Android distributes apps in a bytecode format called DEX. ART runs DEX bytecode directly in an interpreter and also compiles DEX to native ARM64 instructions both at install time (ahead-of-time, or AOT, compilation) and at runtime (just-in-time, or JIT, compilation). Marvin modifies both the interpreter and compiler.

4.3 Marvin Architecture

Marvin has two key components: (1) the *Marvin Kernel* (MK), a modified Android/Linux kernel, and (2) the *Marvin Runtime* (MRT), a modified ART. Most memory management occurs in MRT; it performs working set estimation, ahead-of-time swapping, and bookmarking garbage collection. MK's sole responsibility is to balance memory allocation among apps by deciding when and from which app to reclaim memory.

Marvin performs working set estimation and swapping at object granularity; however, there is no CPU support for object-level access bits and memory faults. As a result, Marvin implements software object access tracking and faulting in the MRT interpreter and compiler. The interpreter marks access bits and checks for swapped-out objects as it runs DEX bytecode; the compiler inserts that functionality as additional ARM64 instructions. Marvin reserves four bytes in each object header and uses them to store swapping metadata and access bits. Implementing these features in software imposes overhead, which we quantify in Section 8. Hardware improvements in future mobile devices could reduce this overhead.

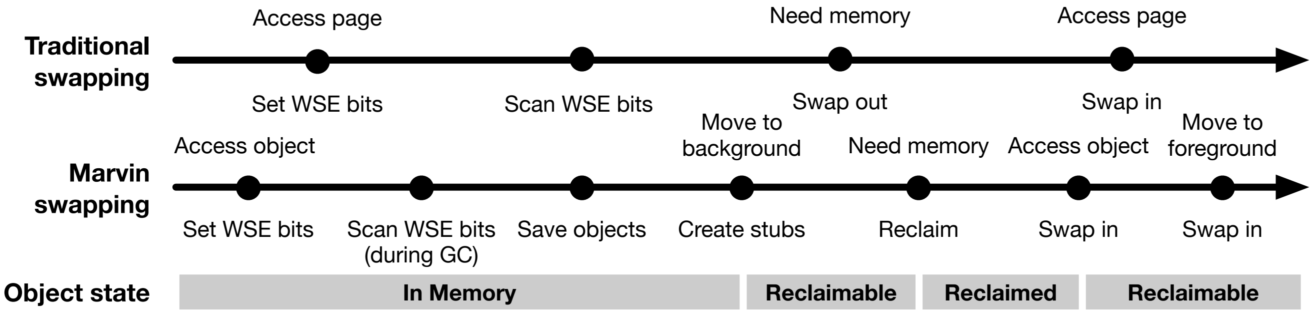


Figure 5: A timeline of actions performed by Marvin's swap mechanism as compared to traditional (e.g., Linux) swap mechanisms. Events are listed above the timeline while Marvin's actions in response are listed below.

4.4 Marvin Memory Management Timeline

Objects managed by Marvin move through several states over time, driven by app behavior and app lifecycle events. Figure 5 illustrates these events and states and compares Marvin's swapping to a traditional swap mechanism. When an app first starts, MRT begins tracking its working set. It identifies objects that are suited for swapping by examining whether they are cold (have not been read or written recently by the app). MRT begins saving checkpoints of those objects to disk in the background. We refer to an object with a saved checkpoint as a *saved* object.

When the app moves from foreground to background, MRT pauses app threads and creates stubs, small proxy objects that add a layer of indirection over swap candidate objects. Stubs ensure that Marvin can intercept accesses to objects and fault them back in, if necessary. Once MRT creates a stub for an object, that object becomes *reclaimable*; the object is still memory-resident, but MK can reclaim its memory at any time. When MK reclaims an object, it enters the *reclaimed* state; only the object's stub remains in memory, and the object's checkpoint will need to be faulted back into memory before the object can be accessed again. The garbage collector uses only the stub and need not fault the object back into memory.

5 Marvin Core Mechanisms

As noted in Section 3, Marvin's key features are ahead-of-time swap, language-aware working set estimation, and bookmarking garbage collection. Designing these features required addressing three challenges: adding a layer of indirection for object references, coordinating between the OS and runtime, and interposing on object accesses. This section describes Marvin's mechanisms for addressing these challenges.

5.1 Stubs for Object Reference Indirection

According to the Java language specification, object references are opaque. However, in practice, object references in the Android Runtime are direct pointers to the heap memory holding the referenced object. This design requires Marvin to inject a layer of indirection to implement features like software object faulting and the bookmarking garbage collector. Marvin creates this layer of indirection using special objects, called *stubs*. Each stub contains a pointer to its underlying object along with a copy of each reference held by the object. All references to an object point instead to its stub, and only the stub holds a pointer to its underlying object. Accessing an object through a stub adds overhead, so Marvin creates stubs only for objects that are cold and at least 2KB in size.

When creating a stub for an object, Marvin moves the object to a separate page-aligned region of memory and then redirects all references to the object to point to its stub instead. These tasks require that all app threads be paused. Therefore, they can be performed more efficiently if Marvin can create stubs for many objects at once, using a single scan of the heap to redirect all affected references. As a result, Marvin periodically executes a heap task that pauses all app threads and creates stubs, and it executes this heap task only when the app is in the background and its threads can be safely paused. Stubs are only created once for each object, so the cost is low, especially because Marvin does not restart apps frequently.

5.2 Reclamation Table for OS-Runtime Coordination

Modern mobile platforms have multicore processors that let system services run concurrently with apps. In this environment, the OS should be able to reclaim memory quickly from a running app without scheduling the app's threads for execution. However, the OS cannot simply seize memory from an app whose threads are not scheduled—a pointer to the memory in question may be present in an app thread's stack or registers, waiting to be used as soon as the thread is scheduled once again. As a result, the OS and runtime need a way to coordinate concurrent accesses to objects so the OS does not try to reclaim one that the runtime is accessing.

Marvin uses a shared-memory *reclamation table* to provide this coordination. MRT populates the table with reclaimable objects, and MK uses it to identify memory to reclaim. Each reclamation table entry is a small, fixed-size data structure that holds the address of an object, its size, a set of flags indicating whether the object is memory-resident and the entry is valid, and a set of bits used for locking by the runtime and OS. To reclaim an object, MK first acquires an exclusive lock on the object's reclamation table entry. Similarly, whenever an app thread prepares to access an object, MRT acquires a shared lock on the reclamation table entry.

The reclamation table is necessary because it provides an agreed-upon location that MK can quickly scan to identify reclaimable objects. If the metadata in the reclamation table were instead stored inside the stubs themselves, then MK would need to scan MRT's entire Java heap to identify reclaimable objects, and stub headers would need to contain a magic number or provide some other way for the kernel to recognize them.

5.3 Object Access Interposition

All of Marvin's features require the runtime to interpose and perform specific tasks whenever an app accesses an object. On every object access, MRT must set read and write bits for working set estimation; check for the presence of a stub and redirect the object access through the stub if necessary; and fault in the object if it has been reclaimed. It must also set a dirty bit whenever an object is modified so the ahead-of-time swap mechanism knows which objects need to be saved, and it must update stubs whenever reference member variables in their corresponding objects change to support the bookmarking garbage collector.

Android apps execute both as DEX bytecode running in an interpreter and as compiled native code running directly on the hardware, and Marvin must interpose on all object accesses in both kinds of code. As a result, Marvin features a set of paired interpreter and compiler modifications that add the required object access interposition. For each additional task performed by the interpreter when it accesses an object, there is a corresponding change to the compiler, adding assembly instructions performing the same task to compiled code.

6 Marvin Memory Management

This section describes how we use Marvin's mechanisms (stubs, the reclamation table, and object access interposition) to design the features that make up Marvin's memory management system.

6.1 Working Set Estimation

MRT performs object-granularity working set estimation by maintaining two access bits in each object header, a read bit and a write bit, and scanning those access bits.

Setting access bits. MRT uses object access interposition to set an object's read and write bits whenever that object is read or written from either interpreted or compiled code. It avoids including garbage collector reads in its working set estimation by setting a flag in the object header when the garbage collector is visiting an object and leaving the read bit untouched if that flag is set.

Access tracking in MRT is performed on a best-effort basis to minimize its overhead: MRT uses non-atomic operations with relaxed memory ordering semantics when setting read and write bits. As a result, concurrent reads and writes to the same object could result in a lost update to one of the access bits. An update to the read bit could also be lost if an app thread reads an object that the garbage collector is processing. These optimizations may decrease swapping performance if the estimated and actual working sets differ significantly, but they do not affect correctness.

Scanning access bits. MRT periodically walks the heap and uses the Clock algorithm [8] to track each object's long-term usage. Each object header holds two four-bit *shift registers*, one for reads and the other for writes. The time between heap walks constitutes an access-tracking "round," and each shift register tracks whether the object was read or written in the last four rounds. During a heap walk, MRT updates an object's shift registers and then clears the object's access bits.

MRT piggybacks off of garbage collection to scan access bits, since GC requires walking the heap anyways. Some of ART's GCs only walk subsets of the heap, so MRT limits its access bit scanning to full-heap collections. It also periodically invokes GC to ensure up-to-date working set estimates in the absence of app activity.

Producing the working set. As MRT walks the heap and scans access bits, it tabulates the app's working set. Our current MRT implementation considers an object part of the working set if it has been written within the last four access-tracking rounds. The precise policy is an implementation detail that can be easily changed.

6.2 Ahead-of-Time Swapping

In Marvin's ahead-of-time swap mechanism, MK reclaims objects and decides which apps to target for reclamation. MRT performs all other functions, including saving object checkpoints to disk, restoring reclaimed objects, and preventing the operating system from reclaiming objects in use by app code.

Saving objects to disk. MRT identifies suitable objects for swapping (i.e., cold objects) using its working set estimation feature, and it proactively saves checkpoints of them to a swap file on disk so they can be reclaimed quickly under memory pressure. MRT saves objects to disk in a periodic heap task that runs on a background thread concurrently with app code.

After app code modifies an object, MRT must save an updated copy of that object to disk. It does not need to save the updated copy immediately as long as it prevents the kernel from reclaiming the object while it is "dirty." To do so, MRT maintains a *dirty bit* in the object header. It uses object access interposition to set this dirty bit whenever app code writes to an object, and its object-saving heap task clears this dirty bit when saving the object to disk. MK checks dirty bits when looking for objects to reclaim and avoids reclaiming dirty objects. MRT and MK use strong memory-ordering semantics when reading and writing the dirty bit to ensure that no modifications to objects are lost.

MRT begins saving swap candidate objects to disk even before those objects have had stubs created for them. Once MRT creates a batch of stubs, those objects become immediately reclaimable without requiring further disk I/O.

Reclaiming objects. MK selects apps to target for reclamation and reclaims objects from the MRT instances corresponding to those apps. It never targets the foreground app, in order to avoid any swapping delays on the foreground app's user interface (UI) thread. After selecting an MRT instance to target, MK scans the MRT instance's reclamation table until it finds an entry for an object that is neither dirty nor locked by the runtime. MK then locks that entry and reclaims the object's pages. It continues scanning the reclamation table and reclaiming objects until it has harvested the desired amount of memory from the MRT instance.

Each MRT instance ensures that MK does not reclaim an object currently being accessed by its app code. To do so, it uses object access interposition to detect whenever a reclaimable object is being read or written, and it locks the object's reclamation table entry before the access and unlocks its entry after the access.

Restoring objects. MRT restores reclaimed objects either eagerly or on-demand. Either way, whenever MRT restores an object, it locks the object's reclamation table entry, copies the saved checkpoint data of the object into memory, and copies any modified references from the object's stub into the object itself. This last step is necessary because references in the stub may have been modified by the garbage collector while the object was not memory-resident.

Marvin's eager object restoration uses app lifecycle information to restore objects before app code needs them. We implemented a simple eager restoration policy, where an MRT instance restores all reclaimed objects when it transitions to the foreground. This policy increases the transition delay in exchange for a guarantee that no swapping delays will block the foreground app's UI thread. Our design is flexible and could support more advanced policies; for instance, the runtime could predict which objects are likely to be touched immediately after a foreground transition and restore those objects first, trading off a shorter pause time in exchange for the risk of user-perceptible stuttering.

If an object has not been eagerly restored, MRT restores it on-demand when app code accesses it, a process that we call *software object faulting*. Whenever app code accesses a reclaimable object, MRT uses object access interposition to check if the object is memory-resident by inspecting a bit in its reclamation table entry. If not, MRT executes an object fault handler that calls into its C++ object restoration function.

6.3 Bookmarking Garbage Collector

A tracing garbage collector touches every object in the heap (or a subset of the heap), causing live objects to be swapped back into memory even if app code is not using them. Marvin's garbage collector avoids touching reclaimed objects by storing an object's references inside its stub and using the stub during the mark phase of garbage collection. Stubs play a similar role as bookmarks in the bookmarking collector [20].

During the mark phase, the garbage collector maintains a *mark stack* and repeatedly pops an object from the mark stack, marks all its references, and pushes those references onto the mark stack. Marvin's garbage collector checks whether an object is a stub when it pops the object from the mark stack; if so, it reads the references off the stub instead of accessing the underlying object.

For the garbage collector to use stubs in place of their objects, MRT must ensure that the stub of a memory-resident object has up-to-date copies of the object's references. It uses object access interposition to update the stub of a reclaimable object whenever Java code modifies one of the object's reference member variables.

MRT must also properly clean up after any saved objects that are freed by the garbage collector. MRT records when saved objects have been freed by the garbage collector, and when the fraction of the swap file consisting of freed objects passes a set threshold (25% in our implementation), it compacts the swap file in a heap task. MRT also cleans up after reclaimable and reclaimed objects by checking whether an object being freed is a stub; if so, it deletes the reclamation table entry corresponding to the stub. If an object is reclaimable, MRT deletes the memory-resident copy of the underlying object; if the object is reclaimed, MRT simply marks its copy in the swap file for deletion without needing to fault it in.

6.4 Design Tradeoffs and Alternatives

By tracking working sets and faulting in objects in software at the runtime level, Marvin achieves a clean design, albeit with some drawbacks. First, Marvin cannot reclaim objects accessed by native libraries: native libraries have no way to detect stubs and no recourse for faulting in reclaimed objects. Second, software working set estimation and object faulting add overhead, particularly to compiled code. We evaluate this overhead in Section 8.

Marvin moves almost all memory management into the runtime because we believe that the runtime's better access to information about app behavior makes it better suited for managing memory. The functionality remaining in the kernel is the minimum required by existing Linux kernel design; if Marvin was built on top of an exokernel [13, 23], it could move even more functionality into the runtime. A variety of other designs are possible that split functionality between the runtime and kernel in different ways.

Kernel-level working set estimation would reduce the overhead of accessing objects, but it would suffer from false sharing if an app's working set is mixed with unused objects across 4KB pages. Faulting in memory at the kernel level would similarly reduce object access overhead but would require more extensive re-design of the runtime garbage collector to avoid unnecessary swapping activity. By tying the granularity of memory management to the size of pages, kernel-level memory management will also become inflexible as large pages become more common and the disparity between object sizes and page sizes widens. In any case, kernel-level memory management would require some sort of ahead-of-time swap mechanism to satisfy the latency requirements of modern mobile platforms (Figure 3), and even adding ahead-of-time swap to the Linux kernel would require significant effort.

Marvin's garbage collector is different from the original bookmarking collector [20] in that it maintains exact reachability information with stubs rather than conservatively storing approximate reachability information. The latter approach requires the garbage collector to perform less work when evicting pages and scanning the heap, but it can result in the heap being needlessly occupied with dead objects.

7 Marvin Prototype

We implemented a prototype of MRT by modifying ART on Android 7.1.1 (which includes Linux 3.18.31). Our implementation includes a modified version of ART's ARM64 compiler, allowing our prototype to support Android devices with 64-bit ARM processors. Our changes to the ART codebase resulted in 3475 additional lines of code [38].

In addition to modifying ART, we made a small modification to the version of OpenJDK included with Android 7.1.1, namely, we added fields to the Object class definition to mirror the bytes added to the object header in ART. We also changed a source file in the Android framework (`ProcessList.java`) to increase a hard-coded limit on the number of concurrently running apps since Marvin is able to run more.

Our experiments require us to manually trigger reclamation, so we did not implement automated reclamation in the MK. However, our MRT implementation includes the reclamation table and performs all operations required to support kernel memory reclamation.

7.1 Object Access Interposition

We implemented MRT's object access interposition by adding specialized functionality to the ART interpreter and compiler. This lets MRT interpose on object accesses from both DEX bytecode running in the interpreter and compiled OAT code running natively. The following section describes in detail how we modified each component.

MRT interpreter. The ART interpreter internally represents each Java object as a C++ *mirror object*, which it manipulates when executing DEX bytecode instructions that read or write an object. The mirror object's type definition includes methods to read or write the data at a given offset within the object's memory footprint, and the interpreter code calls these methods when executing DEX instructions. To add object access interposition to the interpreter, we modified the mirror object methods to implement Marvin's features.

For example, to redirect object accesses through stubs and perform on-demand object faulting, we added a preamble macro to each mirror object method. The preamble first checks if the object is actually a stub. If so, it casts the `this` pointer to a stub, calls a method that locks the stub's reclamation table entry (RTE), checks the RTE's resident bit, and if the resident bit is cleared, calls a method to fault in the object from disk. The preamble then gets the address of the underlying object from the RTE and invokes the mirror object method on the underlying object. Finally, the preamble unlocks the RTE and returns the result of the mirror object method, if any.

ART contains multiple interpreter implementations, and the default is the "mterp interpreter," an interpreter written in assembly. When the mterp interpreter executes DEX instructions that read or write an array, it directly accesses the array's memory, bypassing the mirror object methods. To allow Marvin to interpose on array accesses, we instead use the "switch interpreter," an interpreter written in C++ that calls the mirror object methods when executing array accesses.

MRT compiler. Java code in Android framework libraries and portions of app Java code execute as native code, which is compiled by the ART compiler either statically after installation or dynamically with just-in-time (JIT) compilation. We added object access interposition to this compiled code by modifying the compiler's assembler to generate additional assembly instructions that implement Marvin's features when it performs code generation for object accesses. We used ARM64 devices for testing and evaluation, so we added support for object access interposition to the ARM64 assembler.

Each operation described above for the interpreter's implementation of stub redirection and object faulting has a corresponding block of ARM64 instructions in compiled code. The main difference is that when a stub is detected, the compiled code must explicitly overwrite the register holding the stub's address with the address of the underlying "real object;" it then loads the stub's address back into that register when it is done with the object. In the common case, when an object is not a stub (or when it is, but its underlying "real object" is memory-resident), execution branches past many of the added object-faulting instructions.

7.2 Limitations and Potential Optimizations

Our MRT implementation is a research prototype and has some limitations as a result. One is instability when reclaiming objects from black-box commercial apps. When running apps that we create in Android Studio, MRT reliably and consistently reclaims and restores objects, but when running commercial apps with stub creation and reclamation enabled, it tends to crash. MRT's object access interposition works correctly with commercial apps, so by disabling stub creation and reclamation, we can test its overhead and collect working set data. Our MRT prototype also does not support reclaiming objects with live JNI global references or directly accessing a reclaimable array's memory through JNI.

Our implementation of object access interposition in the MRT compiler is unoptimized, and the per-object-access overhead of compiled code could be reduced with deeper compiler integration. We modified the ARM64 assembler, which translates intermediate representation (IR) instructions to ARM64 binary code. Our implementation generates ARM64 instructions performing object access interposition for every IR instruction that reads or writes an object, even though many of those instructions only need to execute when the object's register allocation begins or ends. An optimized version of the compiler, with modifications at the IR level, could decrease both the execution and size overhead of compiled code.

The MRT compiler has other areas for optimization. For instance, we noticed situations where ARM64 parameter registers (`x0`–`x7` and `d0`–`d7`) appear to be live but are not reported as such by the ART compiler's `LocationSummary` class; therefore, we conservatively save all parameter registers to the stack when performing a procedure call. Saving only live parameter registers would further reduce code size overhead. In addition, the ARM64 assembler makes a maximum of two scratch registers available at any time, which required us to save backpointers and add more instructions to juggle required state among the limited available registers.

8 Evaluation

Our evaluation demonstrated that Marvin successfully met its design goals. It could: (1) quickly reclaim memory on-demand; (2) maintain high memory utilization by sharing memory among apps rather than killing them; and (3) achieve the previous two goals with low overhead and no app changes.

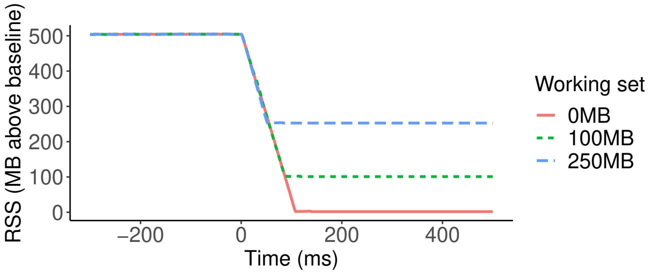


Figure 6: Memory usage as Marvin reclaims memory from a benchmark app with different working set sizes.

8.1 Evaluation Setup

We ran our experiments on a Google Pixel XL smartphone with 4GB RAM and a quad-core Qualcomm Snapdragon 821 CPU. The smartphone ran either the open-source release of Android 7.1.1 (AOSP tag android-7.1.1_r57) or our Marvin implementation based on that release. Both our Marvin implementation and our baseline Android build included a change to the Android framework to increase a hard-coded cap on the number of concurrently running apps.

Our experiments used a mix of synthetic apps for benchmarking and real-world apps. We built several synthetic workloads that simulate various memory footprints and working set sizes to measure their effect on Marvin compared to Android. We also used PCMark for Android, a commercial benchmark app based on real-world apps, to measure Marvin's overhead on real apps [3]. We chose two benchmarks from the test suite (Writing 2.0 and Data Manipulation) since the remaining three test the performance of native libraries.

8.2 Memory Reclamation

Marvin must be able to quickly reclaim memory from running apps when a new or existing app needs to allocate large amounts of memory. Its ahead-of-time swap mechanism ensures that each MRT instance has a pool of clean memory that can be quickly reclaimed without swapping to disk. In this section, we measure the latency of reclaiming memory for apps with different working set sizes. Reclamation latency depends on the app's working set size since Marvin can reclaim more memory from apps with smaller working set sizes.

For our prototype, we use `madvise` to return memory from MRT to the kernel. This design lets us trigger reclamation rather than waiting for memory pressure. Our MRT prototype reclaims memory when an app transitions to the background and then periodically while it is in the background.

Figure 6 shows memory usage over time for apps with a 500MB heap and differing working set sizes. RSS values shown are relative to the RSS reported for a minimal Android app with a single empty Activity (approx. 80MB). At time 0ms, MRT begins to return memory from the app to the OS.

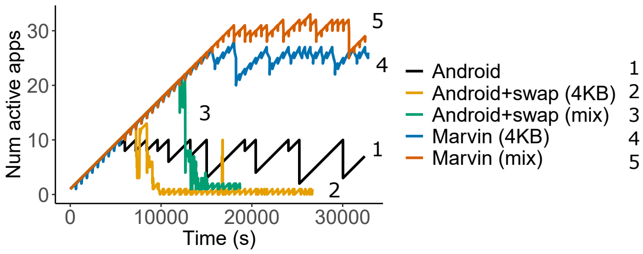


Figure 7: Count of active benchmark app instances over time. Marvin runs more than twice as many apps as regular Android before needing to kill any apps; on Android with a swap file, most apps are alive but inactive due to constant swapping activity.

Marvin returned 250MB of memory in 52ms and 500MB of memory in 108ms. In comparison, as shown in Figure 3, Android with a Linux swap file took nearly 8 seconds to free and allocate 500MB of memory under memory pressure. Using ahead-of-time swap let Marvin reclaim memory over 60x faster than Android with Linux swap could allocate the same amount of memory, allowing Marvin to meet the strict latency requirements of mobile apps.

8.3 Memory Utilization

To demonstrate Marvin's more efficient memory manager, we ran multiple instances of an app with a large memory footprint and a limited working set, and we counted the number of *active* apps that were alive and making progress on their workloads. Each app had a 220MB heap filled with arrays, and it deleted and reallocated 20MB of those arrays every 5 seconds. We used two different heap compositions: one where the apps had heaps filled with 4KB arrays, and one where they had an even mix of 4KB and 1MB arrays (similar to the bimodal distribution of real apps in Figure 1). We consider an app "inactive" if it fails to perform a round of its workload for 20 seconds after the previous round; we consider it "active" once again if it succeeds in performing a round of its workload within 7 seconds of the previous round. We started a new app instance every 10 minutes to give Marvin time to perform background work. For unmodified Android, only the data for the apps with 4KB arrays is shown, because its behavior was nearly identical for the 4KB/1MB mix.

As shown in Figure 7, Marvin ran over 2x as many active apps concurrently as unmodified Android and over 1.5x-2x as Android with a Linux swap file enabled, where swapping left almost all apps unusable as the experiment went on. While baseline Android begins killing apps when physical memory runs out, Android with swap keeps more apps alive. However, without a bookmarking garbage collector, the system experienced constant swapping activity, which prevented most apps from making progress on their workloads. Our experimental runs of Android with a swap file consistently ended early due to the device crashing.

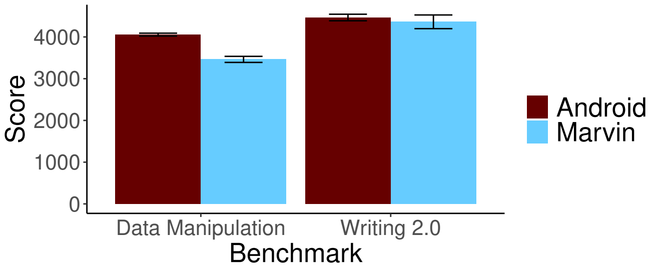


Figure 8: PCMark for Android benchmark results.

Marvin made better use of device memory because it reclaimed unused memory from apps and used its bookmarking garbage collector to avoid touching that unused memory when running garbage collection. While Android only ran 10 apps concurrently, and Android with a swap file only briefly reached a maximum of 13 concurrent apps (4KB arrays) or 20 apps (4KB/1MB mix), Marvin ran 27 apps (4KB arrays) or 30 apps (4KB/1MB mix) concurrently. Marvin's memory reclamation and bookmarking garbage collector let it execute 1.5-2x as many apps concurrently while neither killing apps nor suffering performance degradation.

8.4 Runtime Overhead

While Marvin provides better memory management, it comes with a set of trade-offs. This section quantifies Marvin's four sources of overhead: (1) *execution time overhead* caused by Marvin's object access interposition in compiled OAT code; (2) *increased compiled code size* due to object access interposition; (3) *CPU utilization overhead* caused by Marvin's heap walks for working set estimation; and (4) *faulting overhead* when an app accesses a reclaimed object.

Execution time overhead of object access interposition. Native code produced by the MRT compiler has additional ARM64 instructions to support object access interposition. Some instructions (stub checks, dirty bit updates, and access-tracking bit updates) execute on every object access. Other instructions (indirecting object accesses through stubs and locking RTEs) execute only on accesses to reclaimable objects. We measured the overhead of the added instructions that apply to all object accesses using PCMark for Android, and we used a synthetic benchmark to illustrate the dependence of that overhead on the makeup of application code.

Figure 8 compares the performance of Marvin and unmodified Android on the PCMark benchmarks in our test set. Each bar shows the mean and standard deviation of five runs. We turned off stub creation and swapping when running PCMark, using the benchmarks to measure the overhead of the instructions added to every object access for stub checks and working

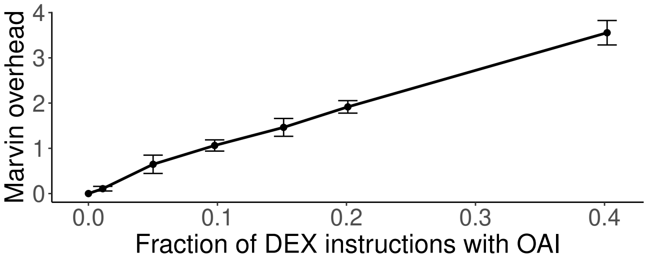


Figure 9: Overhead of Marvin for a synthetic workload with different proportions of object access interposition (OAI). The point (0,0) represents the theoretical scenario of running without any OAI, while other points show experimental results.

set estimation. Marvin's score on the Writing 2.0 benchmark was nearly identical to Android's, and its score on the Data Manipulation benchmark was only 15% lower. These scores show that Marvin's overhead for accessing regular (i.e., non-reclaimable) objects is low for real-world apps.

Figure 9 explores the dependence of Marvin's overhead on the DEX instruction mix of application code. The graph shows Marvin's overhead executing a synthetic workload that performed a tunable proportion of object accesses (array reads and writes) and integer operations (addition and multiplication). Each point represents the mean and standard deviation of Marvin's execution time overhead relative to Android for 40 iterations of the workload. For large proportions of object accesses, Marvin had relatively high overhead (e.g., 350% overhead for 40% object accesses), while for low proportions of object accesses, Marvin's overhead was minimal (e.g., 10% overhead for 1% object accesses). PCMark's 15% overhead indicates that the real app workloads represented by PCMark have low proportions of object accesses.

Although these overheads are already reasonable, they could be improved with optimizations. As noted in Section 7, deeper compiler integration would let Marvin reduce overhead by performing object access interposition less frequently.

Code size overhead of object access interposition. The ARM64 instructions added by Marvin's object access interposition also increase the size of compiled native code. To measure the increase in code size, we compared the compiled Android framework libraries generated by Marvin to the framework libraries on unmodified Android. Marvin increased the total size of the ARM64 framework libraries (in the `/system/framework/arm64` and `/system/framework/oat/arm64` directories on the Android filesystem) from 117 MB to 292 MB. This code size overhead, while relatively high, could be reduced significantly with deeper compiler integration (Section 7).

CPU utilization overhead of heap walks. Our Marvin prototype performs the heap walks required for working set estimation by invoking the concurrent garbage collector and piggybacking off its heap walk. Our prototype performs a heap walk every 5 seconds when an app is in the foreground and every 30 seconds for an app in the background. In theory, this periodic invocation of the garbage collector across multiple MRT instances could add CPU utilization overhead. In practice, when running multiple apps in the background, we found that the difference between Marvin's and unmodified Android's CPU utilization was negligible, likely because GC invocations were so infrequent for background apps.

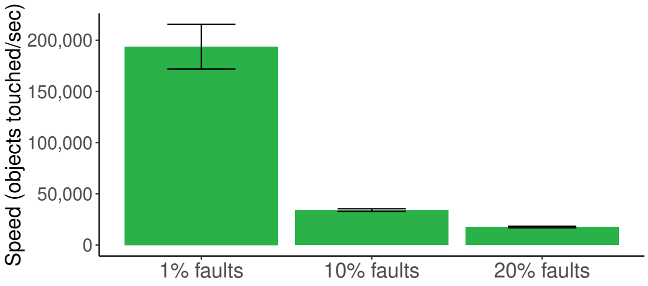


Figure 10: Speed of a benchmark app as it touches objects in its heap with different fractions of reclaimed objects.

Overhead of faulting in objects. When an app first accesses a reclaimed object, Marvin must fault it in from disk, adding significant latency to that initial access. Marvin's default policy eagerly restores all objects when an app moves to the foreground, trading off a longer transition delay for a guarantee that object-faulting latencies will never block the app's UI thread once it is in the foreground. The added transition delay is proportional to the amount of reclaimed memory and the restoration rate. Anecdotally, our prototype restored memory at about 100 MB/s, but we believe that rate could improve to around 260 MB/s with optimization. (The slower rate is the disk read speed of the C++ standard library implementation used by ART, while the faster rate is the speed of the standard library implementation linked by the Android standalone toolchain.) Object faults may occur for background apps, but the Java working sets of apps in the background are generally quite small (less than 4MB for all commercial apps in our test set), so we expect object faulting to happen infrequently in practice.

We nonetheless studied the effect of object faulting on performance, to understand how Marvin would perform in situations where different policies or workloads result in more object faulting. Figure 10 shows the effect of object faulting on a heap-walking benchmark app as it touches different fractions of reclaimed objects. The app looped over a set of 4KB arrays, reading five member variables of each array, and measured the speed of traversing the objects. Each bar shows the mean and standard deviation of five measurements, and the device's disk cache was cleared before each run. As expected, there was an inverse relationship between heap-walking speed and fraction of faults; for instance, speed dropped by 49% as the fraction of faults increased from 10% to 20%.

9 Related Work

Several recent systems provide swapping for mobile platforms but focus on page-granularity rather than object-granularity swapping. SmartSwap [43] predicts which apps are unlikely to be used and swaps out pages from those apps ahead-of-time. A2S [22] takes the opposite approach; it avoids swapping out pages from unused apps, since their pages will be freed anyways when they are terminated. MARS [19] optimizes Linux swapping to improve performance on flash storage devices. It disables garbage collection in background apps and reclaims memory from those apps. DR. Swap [42] uses NVRAM rather than flash storage to store swapped-out pages and satisfies reads by reading directly from NVRAM. Choi et al. [6] improve the performance of an in-memory file system by co-designing the swap mechanism to minimize I/O.

The Linux kernel includes a daemon, *kswapd*, which frees unused pages in the background to maintain a reserve pool of unallocated memory [18]. Like Marvin, kswapd proactively checkpoints unused memory, but unlike Marvin, kswapd reclaims pages when it checkpoints them. As a result, kswapd is limited in how much memory it can checkpoint ahead-of-time—keeping a large proportion of memory checkpointed and reclaimed would make that memory unusable and shrink the device's effective memory footprint.

Liang et al. [24] present FAST, an Android memory management system that modifies kswapd to improve its suitability for Android. FAST changes kswapd to prioritize reclaiming pages from apps in the background. It also identifies a mismatch between the large reclamation sizes of kswapd and the small sizes of typical Android allocations, and it includes a predictor to determine the reclamation size based on workload patterns. Like FAST, Marvin avoids reclaiming memory from the foreground app, but Marvin differs in its runtime-level memory management and its decoupling of checkpointing and reclamation.

A significant body of work examines the issue of providing persistent memory for object-oriented languages [1, 7, 26, 31, 33, 40]. These systems checkpoint objects to disk or non-volatile memory, but they do so to ensure safety in the face of failures rather than swapping out unused memory. As a result, they focus on supporting transactional programming models that provide strong guarantees under failure [7, 31] and on implementing crash-safe garbage collection [7, 40] rather than on maximizing the number of apps that can run concurrently.

SSDAlloc [2] is a persistent memory system that, like Marvin, is motivated by the goal of helping apps with large memory footprints avoid memory pressure. Unlike Marvin's runtime-level object faulting and working set estimation, SSDAlloc allocates objects in separate virtual pages and uses the existing virtual memory system to estimate the working set and trigger its object fault handler.

Like Marvin, the bookmarking collector [20] aims to improve the performance of Java apps in memory-constrained environments. It assumes that the OS uses a traditional page-level swapping mechanism and focuses on letting the garbage collector run without unnecessary swapping. It conservatively stores approximate reachability information (bookmarks) that is used during garbage collection, whereas Marvin stores exact reachability information (stubs). The BMX garbage collector [15] also uses stubs to avoid expensive object accesses, but in the context of a distributed persistent object store.

Other recent work on garbage collection focuses on co-designing the GC and runtime to manage software caches more efficiently [30], co-designing the GC and virtual memory manager to improve performance [41], measuring the effect of GC on scalability [16], and designing GCs or memory managers for domains such as big data systems [17, 29].

With multiple runtimes managing their own memory on top of a single operating system, Android's architecture resembles that of a virtual machine manager, where multiple guest operating systems run on top of a hypervisor. Marvin's runtime–OS cooperation is analogous to that between guest OS and hypervisor in the VMware ESX server [37], which uses a balloon driver to induce guest OSes to reclaim memory.

Wright et al. [39] present a system in which the architecture and Java runtime are co-designed for improved memory access performance. It features hardware modifications that allow an object-addressed CPU cache and an in-cache GC.

10 Conclusion

Users of mobile devices expect to use apps and switch between apps with low latency. As mobile apps have become more memory-hungry, device RAM capacities have not kept pace, and traditional swapping mechanisms cannot meet user latency expectations. Marvin overcomes this challenge with a novel runtime-level swapping mechanism that accurately estimates working sets, moves disk I/O off the allocation critical path, and avoids unnecessary swapping during garbage collection. As our experiments demonstrate, Marvin lets more apps run simultaneously and reclaims memory faster than unmodified Android while adding reasonable overhead. The source code for our Marvin prototype and experiments is available at https://github.com/UWSysLab.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Malte Schwarzkopf, for their insightful feedback. We also thank Sandy Kaplan for editing and writing advice and Ali Razeen for help with Android troubleshooting. This work was funded in part by Futurewei.

References

[1] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, July 1995.

[2] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[3] UL Benchmarks. Pcmark for android. https://benchmarks.ul.com/pcmark-android. Accessed: 2019-1-9.

[4] Kofi Amankwah Boamah. iphone on-board RAM, July 2017. https://www.researchgate.net/figure/Phone-on-board-RAM-From-figure-8-it-is-clear-that-Apple-either-maintains-the-iPhone_fig1_319307164.

[5] Bumptech. Glide v4: Fast and efficient image loading for android. https://bumptech.github.io/glide/. Accessed: 2018-11-28.

[6] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han. In-memory file system with efficient swap support for mobile smart devices. *IEEE Transactions on Consumer Electronics*, 62(3):275–282, August 2016.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[8] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[9] Android Documentation. Activity. https://developer.android.com/reference/android/app/Activity, 2018. Accessed: 2018-11-28.

[10] Android Documentation. Caching bitmaps. https://developer.android.com/topic/performance/graphics/cache-bitmap, 2018. Accessed: 2018-11-30.

[11] Android Documentation. Manage your app's memory. https://developer.android.com/topic/performance/memory, 2018. Accessed: 2018-11-28.

[12] Android Documentation. Saving ui states. https://developer.android.com/topic/libraries/architecture/saving-states, 2018. Accessed: 2018-11-28.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[14] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 110–122, 2018.

[15] Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI '94)*, pages 229–241, Monterey CA, USA, United States, 1994.

[16] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*, 2011.

[17] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015.

[18] Mel Gorman. An investigation into the theoretical foundations and implementation of the linux virtual memory manager, 2003.

[19] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, and Weimin Zheng. Mars: Mobile application relaunching speed-up through flash-aware page swapping. *IEEE Transactions on Computers*, 65(3):916 – 928, March 2016.

[20] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, 2005.

[21] Tyler Kieft. Building a better instagram app for android. https://instagram-engineering.com/building-a-better-instagram-app-for-android-c08f973662b, 2014. Accessed: 2018-11-9.

[22] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s):182:1–182:19, September 2017.

[23] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[24] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020.

[25] Michelle Meyers. Android inches ahead of windows as most popular os. CNET, April 2017. https://www.cnet.com/news/android-most-popular-os-beats-windows-statcounter/.

[26] J. Eliot B. Moss. Design of the mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[27] Mike Nakhimovich. Improving startup time in the nytimes android app. https://open.blogs.nytimes.com/2016/02/11/improving-startup-time-in-the-nytimes-android-app/, 2016. Accessed: 2018-11-9.

[28] Randy Nelson. The size of iphone's top apps has increased by 1,000% in four years. Sensor Tower, Jun 2017. https://sensortower.com/blog/ios-app-size-growth.

[29] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[30] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. Prioritized garbage collection: Explicit gc support for software caches. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 695–710, New York, NY, USA, 2016. ACM.

[31] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 161–174, New York, NY, USA, 1993. ACM.

[32] Anshu Rustagi. How we improved our android app "cold start" time by 28%. https://redfin.engineering/how-we-improved-our-android-app-cold-start-time-by-28-a722e231314a, 2018. Accessed: 2018-11-9.

[33] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 11–33, London, 1993. Springer London.

[34] Facebook Open Source. Fresco. https://frescolib.org/. Accessed: 2018-11-28.

[35] StackExchange. Creating and enabling an internal storage swap partition on rooted android kitkat. https://android.stackexchange.com/a/89030. Accessed: 2019-4-4.

[36] StackOverflow. ios app maximum memory budget. https://stackoverflow.com/a/15200855. Accessed: 2019-1-9.

[37] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[38] David A. Wheeler. SLOCCount, 2013. http://www.dwheeler.com/sloccount/.

[39] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2005.

[40] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[41] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[42] K. Zhong, X. Zhu, T. Wang, D. Zhang, X. Luo, D. Liu, W. Liu, and E. H.-M. Sha. Dr. swap: Energy-efficient paging for smartphones. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 81–86, Aug 2014.

[43] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. Smartswap: High-performance and user experience

friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 22:1–22:6, New York, NY, USA, 2017. ACM.

Retwork: Exploring Reader Network with a COTS RFID System

Jia Liu[1] Xingyu Chen[1] Shigang Chen[2] Wei Wang[1] Dong Jiang[1] Lijun Chen[1]
[1]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
[2]*Department of Computer & Information Science & Engineering, University of Florida, USA*

Abstract

Radio frequency identification has been gaining popularity in a variety of applications from shipping and transportation to retail industry and logistics management. With a limited reader-tag communication range, multiple readers (or reader antennas) must be used to provide full coverage to any deployment area beyond a few meters across. However, reader contention can seriously degrade the performance of the system or even block out some tags from being read. Most prior work on this problem requires hardware and protocol support that is incompatible with the EPC Gen2 standard. Moreover, they assume the knowledge of a reader network that precisely describes the contention relationship among all readers, but the efficient acquisition of the reader network in a practical system with commercial-off-the-shelf (COTS) tags is an open problem. This study fills the gap by proposing a novel protocol *Retwork*, which works under the limitations imposed by commercial Gen2-compatible tags and identifies all possible reader contentions efficiently through careful protocol design that exploits the flag-setting capability of these tags. We have implemented a prototype with 8,000 commercial tags. Extensive experiments demonstrate that *Retwork* can reduce communication overhead by an order of magnitude, in comparison to an alternative solution.

1 Introduction

Radio frequency identification (RFID) has been gaining popularity in a variety of pervasive applications, including library inventory [7,13,16,25,26], warehouse control [6,17,21,22,24,38–41], supply chain management [12,15,19,20,23], and object tracking [9,18,29,31–34,36]. Given that tags use backscattering to communicate with readers, the communication distance between a reader and a tag is limited to a few meters. In a deployment (e.g., a retail store or a warehouse) that goes beyond the communication range of a single reader antenna, multiple reader antennas (referred to simply as *readers* for convenience) must be used to cover the whole area. If the readers take turn to communicate with the tags in their respective interrogation zones, then this condition will not be time efficient for inventory operations. On the contrary, if they operate simultaneously, a complex situation of collisions, where tags in overlapped areas will be left unread, may be created. To solve this dilemma, research has been trying to find solutions to properly schedule readers so that only those that do not collide will be active at any time.

This reader scheduling is built upon the knowledge of *reader network*, which is a graph that depicts the contention relationship among the readers and underlies many multi-reader protocols [5,8,10,14,27,28,30,35,37,42]. However, in practice, a reader can hardly know the size of its own interrogation zone, which takes an irregular shape that is difficult to determine due to directivity of reader antenna and environment reflection. More difficulty is to determine whether any two readers contend, which happens when their interrogation zones overlap and at least one tag in the overlapped area exists. To make reader scheduling practical, in this paper, we work under the limitations of commercial Gen2-compatible RFID systems and propose a solution to determine their reader network, without any modification to tags or reader-tag communication protocols, and without any assumption of pre-knowledge about the shape and size of any reader's interrogation zone. One naive solution is to activate the readers in sequence, one at a time, to collect tag IDs in its zone, and then compare the tag sets of any two readers to see if the intersection is empty. If it is, then no contention exists between the two readers; if not, there is a contention link between them in the network. However, this serialized approach is inefficient and already collects all tag IDs, which makes deriving the reader network unnecessary.

This paper proposes a new protocol called *Retwork* that efficiently determines the reader network of a large RFID system, with two key advantages. First, it avoids the need to perform inventory over the entire tag set by the native solution that activates one reader at a time (which reads its tags, one at a time). Second, it is completely compatible with the worldwide standard of EPC Gen2 [4], allowing the new protocol to be deployed in commercial systems. The idea behind *Retwork* is not to read all tags and compare the readers' tag sets, but to make each reader broadcast certain information to tags in its zone. The Gen2 standard does not allow us to write a reader's ID to all tags in its zone at once. Had this been supported, after all readers did that, tags would know whether they are in the overlapped areas of multiple readers and would then report that to their readers. Fortunately, according to the Gen2 standard, a reader's transmission can flip certain flags in the memory of all tags that receive the transmission. With a careful design, we show that these flags can be exploited to fully support identification of all contention relationship among the readers. Our protocol only requires each reader to transmit a few Gen2 commands that trick its tags to flip their flags in a certain way. Thus, after all readers transmit, tags in overlapped areas will have their flags set differently from other tags. Tags will signal the readers for contention relationship without having to deliver their IDs to the readers. The execution time of *Retwork* is a function of the number of readers, instead of the number of tags, which is much larger in practical systems. The major contributions of this study are listed below.

- We propose an efficient solution *Retwork* to the practically important problem of identifying the contention relationship among multiple readers in a large RFID system, which underlies a majority of multi-reader protocols.
- Our protocol exploits the flag-setting capability in Gen2. With a carefully-designed series of flag-flipping operations, our protocol can classify tags into groups: those under reader contention and those free of contention.
- We implement a prototype of *Retwork* with 8,000 commodity RFID tags. Extensive experiments show that it boosts the read throughput of the system and thus cuts the inventory time by an order of magnitude.

2 Problem Formulation

An RFID system generally consists of a large number of tags and multiple readers. Each tag is attached to an object to exclusively indicate the associated object. The tag set is denoted by $\Gamma = \{t_1, t_2, ..., t_n\}$. A reader is surrounded by a finite space within which it can communicate with tags. This space is referred to as the *interrogation zone* (or *read zone*) of that reader. In a multi-reader RFID system, the layout of readers constitutes a *reader network*, which is represented by a graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, ..., v_m\}$ is the set of vertices (readers) and E is the set of edges (contention links). An edge $(v_i, v_j) \in E$ exists between the reader v_i and the reader v_j if and only if at least one tag $t_k \in \Gamma$ is located at the overlapped interrogation zone covered by both readers. At this point, these two readers are called *neighbors* or *adjacent nodes*, which may incur collision if they communicate concurrently. The tags within the overlapped zone are referred to as *contentious tags*. Notably, forming an edge requires two necessary conditions: overlapped read zone and contentious tags. Two readers with the same read zone are still treated as collision-free if no tags reside in such a zone. This is reasonable because no contention will happen even if the two readers run in parallel. Note that, various factors, such as the reader planning, multi-path effects, and material of tagged objects, greatly affect a reader's signals and thus make the shape of its interrogation zone irregular. Our protocol design is robust to any kinds of RFID systems, regardless of the shape of the interrogation zone.

3 Tag Inventory

Exploring the reader network is essentially to check whether a contention link exists between any pair of readers. An intuitive solution is to conduct tag inventory reader by reader over the whole tag set. In particular, each reader individually queries the tag subpopulation in the field of view. Upon receiving a query request, all tags report their tag IDs to the reader by running the Gen2 protocol. To avoid reader collision, all readers need to execute the tag inventory sequentially rather than concurrently. That is because these readers do not have any prior knowledge on the reader network; a collision is very likely to take place when concurrent inventory is conducted. This blocks out some tags in the overlapped zone from being read. After one-round inventory by all readers, each reader can learn its neighbors by comparing its own tag list with others'. If two readers share a common tag subset, an edge must exist between them; otherwise, they are conflict-free. By checking all pairs of readers, the reader network $\mathcal{G} = (V, E)$ is formed finally. This solution is foolproof but suffers from high latency. The reason is that all tags have to transmit their long tag IDs to readers, resulting in at least $n \times t_{id}$ time overhead, where n is the number of tags and t_{id} is the time delay for transmitting a tag ID. This process is extremely time consuming, especially in a large RFID system.

4 Retwork

4.1 Basic Idea

The basic idea of *Retwork* is piggybacking some payload in a reader's instruction via one-to-many broadcasting over the air and taking a few tag replies instead of all as the indicator to obtain the link information between readers. By this means, most tag inventories are avoided and the identification time of reader network is determined by only the small number m of readers rather than the number n of tags ($m \ll n$), greatly improving the protocol efficiency. Following this idea, we propose *Retwork* in embryo, which gives us a clue to the protocol design and reveals the key hurdle that limits the implementation of *Retwork* on Gen2. It consists of two phases: *over-the-air writing* and *selective reading*. The former is to tell each tag in which readers' interrogation zones the tag resides via one-to-many broadcast. The latter chooses a specific tag subset to reply. By checking the tag responses, the reader is able to learn whether two readers conflict.

Over-the-air Writing. This phase is composed of m time slots, where m is the number of readers. Each reader is assigned an exclusive slot and scheduled to transmit its reader index in that slot. Without loss of generality, we suppose that the reader v_i is assigned to the i-th slot. The reader v_i broadcasts its index i to all tags in its vicinity during the i-th slot. Clearly, only the tags within the reader's interrogation zone can hear this broadcast. By recording reader indices, a tag knows at which readers' interrogation zones it is located. More specifically, each tag holds an m-bit indicator vector in its memory, which is denoted by I and initializes to zeros at first. Once a tag receives an index of value i, it sets the i-th bit of the vector to '1', that is, $I[i] = 1$, which indicates that this tag is under v_i's coverage. A contention link between v_i and v_j is formed if two bits $I[i]$ and $I[j]$ meet $I[i] = 1$ and $I[j] = 1$, where $1 \leq i < j \leq m$. By checking all indicator vectors and converging these pieces of information together, we can obtain the reader network $\mathcal{G}(V,E)$. However, although tags know all of this, the readers do not. We next introduce the second phase that aims to extract the contention information from tags and shed light on the reader network.

Selective Reading. Directly collecting each tag's indicator vector can obtain the reader network but suffers from long time delay as tag inventory. To improve time efficiency, the reader chooses only a few tags to reply each time and removes most of dispensable memory accesses. In particular, each reader in turn checks whether it conflicts with others. Consider any one reader v_i, $1 \leq i \leq m$. To obtain the link status between v_i and v_j $(j > i)$, the reader v_i first selects a specific subset of tags with indicator vectors that satisfy $I[j] = 1$. If the reader v_i detects any tags in the field of view, then v_i and v_j are neighbors for sure. Otherwise, no response from tags means no tags hold $I[j] = 1$, which further indicates that v_i and v_j are conflict-free.

Given the use of over-the-air writing and selective reading, the execution time relates to only the number m of readers, regardless of the number n of tags. Since m is much smaller than n in practice, the proposed solution greatly improves the time efficiency compared with tag inventory (in §3) that needs more than n tag collections.

4.2 Challenge in Implementation

Consider the above two phases. The selective reading can be well supported by the *Select* command specified in Gen2 (see §4.3). Over-the-air writing, however, needs the reader to write a group of tags in its vicinity via one instruction. We refer to this one-to-many write operation as *BlastWrite*, which is however out of the scope of Gen2 that specifies the reader has to perform memory access on one tag at a time. This condition makes building indicator vectors roll back to one-to-one transmission again.

4.3 EPCglobal Gen2 Protocol

The EPCglobal Gen2 protocol [4] defines the physical interactions and logical operating procedures between readers and tags. We highlight two functions that we will use on *Retwork* shortly later.

Select Command. *Select* is a mandatory command that can assert or deassert a tag's selected (SL) flag, or set a tag's inventoried flag to either A or B. These flags are used to determine whether a tag may respond to a reader, which is the key to identify the reader network (details are given in §4.4). *Select* comprises six mandatory fields.

- *Target.* It indicates the object that *Select* will operate, which is either a tag's SL flag or an inventoried flag in any one of four sessions. Sessions are specified by Gen2 to fit the case of exclusive reading amongst multiple readers.
- *Action.* This field elicits the action to be taken by a tag. Eight actions are available, where matching and not-matching tags assert or dessert their SL flags, or set their inventoried flags to A or B. By combining *Target* and *Action*, the reader is able to modify a specific flag. For example, a matching tag's inventoried flag in session 2 will be set to A when *Target*$= 010_2$ and *Action*$= 000_2$.
- *MemBank*, *Pointer*, *Length*, *Mask*. The four fields jointly determine which tags are matched for *Action*. *MemBank* specifies the memory bank. *Pointer* indicates the starting position. *Length* determines the length of *Mask*, which is a customized bit string according to upper application requirements. If *Mask* is the same as the string that begins at *Pointer* and ends *length* bits later in the memory of *MemBank*, then the corresponding tag is matched.

Query Command. After *Select*, *Query* initiates and specifies a new inventory round over the tag subpopulation chosen by *Select*. In the inventory round, the reader will play out a frame that consists of a group of time slots. Each selected tag randomly picks one of these time slots and transmits its tag ID to the reader in that slot. A tag inventory may need to execute several inventory rounds and is finished after all selected tags successfully reply to the reader. *Query* command includes three fields that we concern.

- *Sel.* This field consists of two bits that determine which tags respond to *Query*: 00_2 and 01_2 indicate all matching tags in the previous *Select* command; 10_2 indicates tags with deasserted SL flag; 11_2 indicates tags with asserted SL flag.
- *Session.* It selects a session for the subsequent inventory round. Gen2 requires readers and tags to provide four sessions (denoted as S0, S1, S2, and S3). Tags in one of these sessions shall neither use nor modify an inventoried flag for a different session. This way allows two or more readers to

use different sessions to independently inventory a common tag population (in different time slots).

• *Target.* This field chooses whether tags with inventoried flag of A or B participate in the upcoming inventory round, where 0 indicates A and 1 indicates B. Tags may invert their inventoried flags from A to B (or vice versa) after being successfully queried.

4.4 Design of Retwork

Although *BlastWrite* is not supported by Gen2, a reader's command (e.g., *Select* and *Query*) can be transmitted to all tags simultaneously through one-to-many broadcast. If we can piggyback some useful information in a reader's command such that all tags' memories are updated in the meanwhile, then an equivalent mimic of *BlastWrite* might be implemented on Gen2. An indicator flag (inventoried flag or SL flag) can make this condition possible because all tags' indicator flags can be set by a single reader command. Below, we detail the use of reader commands together with the Gen2-compatible indicator flag to obtain the reader network. For ease of presentation, we choose the inventoried flag in session 2 (S2) as the vehicle to show how *Retwork* works; other indicator flags can also be adopted similarly.

4.4.1 Detection of Contention Link

Consider any two readers v_i and v_j, $1 \leq i < j \leq m$. The contention link between v_i and v_j can be determined by the following three steps: (i) the reader v_i broadcasts a *Select* command to set all inventoried flags of the tag set in its vicinity to A; (ii) the reader v_j performs the similar operation that sets the inventoried flags to B; (iii) the reader v_i issues a *Query* command and executes the tag inventory on the subset of tags with inventoried flags of B. If v_i and v_j are neighbors, the tags in the overlapped zone must be set to B and v_i can get replies from these tags. Otherwise, none of tags in the field of view of v_i is set to B and nothing will be received. Accordingly, by checking tag replies, v_i can learn whether it conflicts with v_j. Next, we elaborate the way to achieve the above function with Gen2-compatible *Select* and *Query*. A *Select* command is denoted by:

$$\mathcal{S}(\underbrace{t}_{Target},\overbrace{a}^{Action},\underbrace{b}_{MemBank},\overbrace{p}^{Pointer},\underbrace{l}_{Length},\overbrace{k}^{Mask}), \tag{1}$$

with the fields of *Target* (t), *Action* (a), *MemBank* (b), *Pointer* (p), *Length* (l), and *Mask* (k). To set all tags' inventoried flags (in S2) to A, the reader needs to broadcast:

$$Flag = A : \mathcal{S}(2,0,1,0,0,0),$$

where $t = 2$ (010_2) means the operating object is set to the inventoried flag in session 2 (S2), $a = 0$ indicates that the inventoried flags of matching tags will be set to A while those of not-matching will be set to B, and $(b, p, l, k) = (1,0,0,0)$ means all tags within the coverage are selected (matching). Similarly, to set the inventoried flag to B, the reader only needs to carry out the same *Select* except that the value of *Action* field is altered to 100_2 ($a = 4$). Thus, we have:

$$Flag = B : \mathcal{S}(2,4,1,0,0,0).$$

After *Select*, a *Query* is needed for inventory:

$$Q(\underbrace{e}_{Sel},\overbrace{s}^{Session},\underbrace{g}_{Target}). \tag{2}$$

To inventory all tags with inventoried flags in S2 of B, the query command shall be $Q(0,2,1)$, where *Sel* (e), *Session* (s), and *Target* (g) are 00_2, 10_2, and 1, respectively. By combining the *Select* and *Query* together, we obtain the instructions for detecting the contention link between v_i and v_j.

$$\begin{aligned} &\textcircled{1}\ v_i:\ \mathcal{S}(2,0,1,0,0,0)\\ &\textcircled{2}\ v_j:\ \mathcal{S}(2,4,1,0,0,0)\\ &\textcircled{3}\ v_i:\ Q(0,2,1). \end{aligned} \tag{3}$$

With these commands, the contentious tags (if appropriate) in the overlapped zone between v_i and v_j are isolated from others; the reader v_i can check the existence or absence of these tags by executing a short inventory round. Compared with the basic tag inventory over the entire tag set, this way avoids a great number of tag memory accesses, regardless of the number of tags.

4.4.2 Identification of Reader Network

The findings above indicate that an intuitive solution to identifying the reader network is to detect each pair of readers with the instructions of (3) and later draw the reader graph with the identified contention relationships. This method is far superior to the inventory-based solution as most tag inventories can be avoided. However, this one-pair-at-a-time scheme suffers from repetitive transmissions for setting inventoried flags, increasing the communication overhead. Take the reader v_1 for example. To get the contention relationships between v_1 and other readers, the reader v_1 needs to execute the operation of setting A (① in (3)) $m-1$ times. If this repetition can be avoided, then a great deal of communication overhead will be saved, which improves the protocol performance. To this end, we propose a scheduling policy that tries to reduce the number of broadcasts of instructions and improve the global time efficiency for obtaining the reader network.

The basic idea is that, instead of solely checking each pair of readers, we identify a group of contention relationships by simultaneously taking multiple readers into account. To identify the entire reader network, $m-1$ identification rounds are needed (m is the number of readers). In each round, we select a reader and identify the contention relationships between this reader and others. More specifically, all readers except for v_1 initially set the inventoried flags to A. Thereafter, the identification starts, round by round. In the i-th round ($1 \leq i < m$), we select the reader v_i and check whether it conflicts with other readers v_j ($j > i$) via three steps. (i) The reader v_i solely updates the flag to B. (ii) Each reader v_j in turn queries the tags with the flag equal to B. If any v_j conflicts with v_i, then the tags in the overlapped zone must be set to B in the first step and v_j can obtain the reply from these tags. Otherwise, no tags in the field of view of v_j reply. By this means, all the contention relationships between v_i and v_j are identified. Compared with the one-pair-at-a-time scheme, the reader v_i executes the flag setting only once rather than $m-1$ times, greatly saving the communication overhead. (iii) In the last step, we reset all flags under v_j's coverage to A again, which would be the input of the next round. For this purpose, one broadcast of $\mathit{Flag} = A$ by v_i is sufficient; no need for all readers v_j to do so. That is because the flags that transform from A to B are due to the flag setting by v_i, that is, the tags with flag B must be under v_i's coverage. After the three steps, the current round terminates and we move to the next one. This process repeats round by round until the global reader network is identified.

4.5 Time Efficiency & Improvement

Now, we discuss the execution time of *Retwork*. As aforementioned, $m-1$ identification rounds need to be run to obtain the reader network. Consider the i-th round, $1 \leq i < m$. It consists of two *Select* commands (sent by v_i) and $m-i$ *Query* commands (issued by each reader v_j, $j > i$). Therefore, the execution time of the i-th round is $2t_s + (m-i)(t_q + t_v)$, where t_s, t_q, and t_v are the time intervals of a *Select* command, a *Query* command, and an inventory round that checks whether requested tags exist, respectively. By considering $m-1$ rounds together with initial $m-1$ flag settings, we have the execution time T of *Retwork*:

$$T = \frac{m(m-1)}{2}(t_q + t_v) + 3(m-1)t_s. \qquad (4)$$

This execution time is determined by only the number of readers once the transmission rate of reader-tag communication is fixed, regardless of the large number n of tags. This way is a great performance boost compared with tag inventory given that m is much smaller than n. For example, in a practical scenario (e.g., warehouse or library), a reader usually covers more than 1000 passive tags, i.e., $m \leq 0.001n$. Below, we propose two schemes that further improve the time efficiency of *Retwork*. First, Gen2 allows a tag to send a truncated reply (i.e., a portion of EPC) by enabling *Truncate* field of the *Select* command. This way will help a tag reduce the transmission of the 96-bit EPC to only a single bit, which reduces the communication delay of tag replies. Second, if the range limit of RFID readers is considered, we do not need to check two readers beyond the communication range, which sharply reduces the number of contention detection.

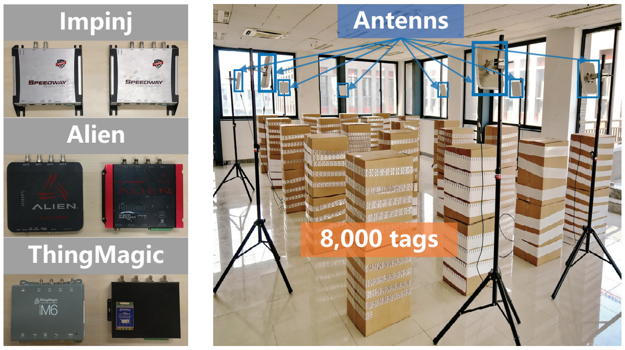


Figure 1: Experimental setup.

5 Evaluation

We evaluate *Retwork* using COTS RFID readers and tags. Six models of UHF RFID readers from three experienced suppliers of RFID products are used in our experiments: ALR-F800 and 9900+ from Alien Inc. [1], R220 and R420 from Impinj [2], Mercury6 and M6e from ThingMagic [3]. Each reader is connected to a directional antenna Larid S9028PCR [11] that is with 8.5 dBic gain and operates at around 900 MHz. To better mimic a real RFID system and extensively study the performance of *Retwork* in practice, we use a total of 8,000 commodity tags in our experiments. As shown in Fig. 1, these tags are densely attached to 40 cartons, each of which contains approximately 200 tags. The readers are deployed with three kinds of densities, namely, sparse (four readers), moderate (six readers), dense (eight readers), to cover a desired area of 12m×8m. The sparse fits for the scenarios of dock door or conveyor belt; the moderate is used for the case of laboratories or office; the dense is suitable for the case of libraries or shopping malls.

5.1 Identification Accuracy

Retwork examines each contention link between two readers by broadcasting *Select* and *Query* commands. Due to the manner of one-to-many transmission, *Retwork* might incur some false positives or false negatives. A false positive is a result that indicates a contention link exists, when it does not indeed. On the contrary, a false negative indicates that two readers do not conflict, while in fact they do. We next study the identification accuracy of *Retwork*, which is measured by false positive ratio (FPR) and false negative ratio (FNR). The ground truth is obtained by executing the inventory-based solution: each reader conducts the tag inventory in sequence and later compares its own tag list with others'; if two readers share a common tag subset, then they conflict. Fig. 2 shows FPR and FNR in three scenarios of different reader densities.

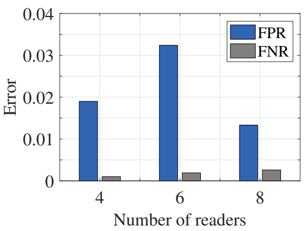
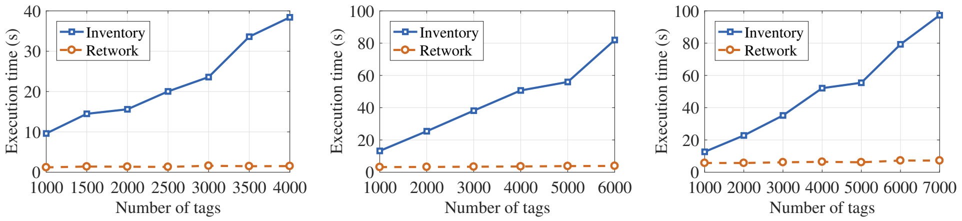


Figure 2: Accuracy.



(a) Sparse.

(b) Moderate.

(c) Dense.

Figure 3: Time comparison between *Retwork* and tag inventory.

As observed, FPR and FNR are bounded within a low level of errors. Although FPR is relatively larger than FNR, false positives slightly negatively affect the functions of multi-reader protocols. Instead, false negatives might incur some errors: two neighbor readers work concurrently. However, the FNR is 0.3%, which is very small for most applications. We can further decrease FNR by running *Retwork* multiple times if desired.

5.2 Time Efficiency

The usage of inventoried flags and reader commands avoids most tag inventories and thus lowers the identification latency of the reader network. This is where *Retwork* shines. Now, we study the time efficiency of *Retwork* over the three reader scenarios, where tag inventory in §3 is taken as the baseline for comparison. Fig. 3 plots the execution time of *Retwork* and tag inventory with respect to the number of unique tags that have been read. As observed, *Retwork* is far superior to tag inventory under difference cases. For example, in the moderate case (six readers) with 5,000 tags (Fig. 3(b)), *Retwork* reduces the execution time from 55.9s to only 3.9s, producing a 14.7× performance gain. Given a reader deployment, the execution time nearly remains stable, regardless of the number of tags. This result is consistent with our previous belief in the protocol design. With the increase in the number of readers, the number of detection units required by *Retwork* increases, as well as the execution time. Notably, the execution time here is the worst case of *Retwork*. If the truncated reply and range limit are taken into account (see §4.5), then the performance of *Retwork* will be further improved. The execution time of tag inventory does not see a linear rise over the number of tags because the number of tags in the overlapped read zones increases correspondingly.

6 Related Work

In a multi-reader RFID system, reader collisions occur frequently and inevitably, which impairs the read throughout and leads to misreads. Research has been trying to find solutions to properly schedule readers to ensure that only those that do not collide will be active at any time. Colorwave [30] is one of the first work to address reader collision. It randomly colors readers such that each pair of interfering readers have different colors. In AcoRAS [8], readers are assigned colors by a central server following the build of a minimum independent set. Season [37] proposes a scheme of reader collaboration to improve the time efficiency of tag inventory by using two steps: shelving the collisions and identifying the tags that do not involve reader collisions; performing a joint identification, in which adjacent readers collaboratively identify the contentious tags. Liu et. al [14] propose a maximum-weight-independent-set-based approximation algorithm to address the problem of reader-coverage collision avoidance: activating readers and adjusting their interrogation ranges to cover maximum tags without collisions subject to the limited number of tags read by a reader. In spite of the advancement, the reader scheduling largely depends on the knowledge of reader network, which is a graph that depicts the contention relationship among the readers and underlies many prior multi-reader protocols. Obtaining the reader network, however, is no picnic in practice.

7 Conclusion

In this work, we investigate the fundamental problem of exploring reader network, which is vital to reader scheduling and underlies many anti-collision protocols in a multi-reader RFID system. A Gen2-compatible protocol *Retwork* is proposed to identify reader network on COTS devices. By exploiting flag-setting capability of commercial tags, *Retwork* avoids most tag inventories and improves time efficiency. Extensive experiments show that *Retwork* can reduce the execution time by an order of magnitude.

Acknowledgments

We would like to thank our shepherd, Prof. Lin Zhong, and the anonymous reviewers for their valuable feedback. This work is supported by National Natural Science Foundation of China (Nos. 61702257 and 61771236), Natural Science Foundation of Jiangsu Province (BK20170648), Jiangsu Key R&D Plan (Industry Foresight and Common Key Technology, BE2017154), US National Science Foundation CNS-1718708, Fundamental Research Funds for the Central Universities (14380066), and Collaborative Innovation Center of Novel Software Technology and Industrialization. Jia Liu and Lijun Chen are the corresponding authors.

References

[1] Alien Technology. http:// www. alientechnology. com.

[2] Impinj Inc. http:// www. impinj. com.

[3] Thingmagic. http:// www. thingmagic. com.

[4] *GS1 EPCglobal. EPC radio-frequency identity protocols generation-2 UHF RFID version 2.0.1*, 2015.

[5] Maurizio A. Bonuccelli and Francesca Martelli. A very fast tags polling protocol for single and multiple readers RFID systems, and its applications. *Ad Hoc Networks*, 71:14–30, 2018.

[6] Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID counting protocols. In *Proc. of ACM MobiCom*, pages 291–302, 2013.

[7] Isaac Ehrenberg, Christian Floerkemeier, and Sanjay Sarma. Inventory management with an RFID-equipped mobile robot. In *Proc. of IEEE CASE*, pages 1020–1026, 2007.

[8] Essia Hamouda, Nathalie Mitton, and David Simplot-Ryl. Reader anti-collision in dense RFID networks with mobile tags. In *Proc. of IEEE RFID-TA*, pages 327–334, 2011.

[9] Jinsong Han, Chen Qian, Xing Wang, Dan Ma, Jizhong Zhao, Pengfeng Zhang, Wei Xi, and Zhiping Jiang. Twins: Device-free object tracking using passive tags. In *Proc. of IEEE INFOCOM*, pages 469–476, 2014.

[10] Nikolaos Konstantinou. Expowave: An RFID anti-collision algorithm for dense and lively environments. *IEEE Transactions on Communications*, 60(2):352–356, 2011.

[11] Larid. S9028PCL. https:// www. lairdtech. com/ products/ s9028pcl .

[12] Chun-Hee Lee and Chin-Wan Chung. RFID data processing in supply chain management using a path encoding scheme. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):742–758, 2011.

[13] Renjun Li, Zhiyong Huang, Ernest Kurniawan, and Chin Keong Ho. AuRoSS: an autonomous robotic shelf scanning system. In *Proc. of IEEE/RSJ IROS*, pages 6100–6105, 2015.

[14] Bing-Hong Liu, Ngoc-Tu Nguyen, Van-Trung Pham, and Yu-Huan Yeh. A maximum-weight-independent-set-based algorithm for reader-coverage collision avoidance arrangement in RFID networks. *IEEE Sensors Journal*, 16(5):1342–1350, 2016.

[15] Jia Liu, Bin Xiao, Kai Bu, and Lijun Chen. Efficient distributed query processing in large RFID-enabled supply chains. In *Proc. of IEEE INFOCOM*, pages 163–171, 2014.

[16] Jia Liu, Feng Zhu, Yanyan Wang, Xia Wang, Qingfeng Pan, and Lijun Chen. RF-Scanner: Shelf scanning with robot-assisted RFID systems. In *Proc. of IEEE INFOCOM*, pages 1–9, 2017.

[17] Xuan Liu, Bin Xiao, Feng Zhu, and Shigeng Zhang. Let's work together: Fast tag identification by interference elimination for multiple RFID readers. In *Proc. of IEEE ICNP*, pages 1–10, 2016.

[18] Jiaqing Luo and Kang G Shin. Detecting misplaced RFID tags on static shelved items. In *Proc. of ACM MobiSys*, pages 378–390, 2019.

[19] Saiyu Qi, Yuanqing Zheng, Xiaofeng Chen, Jianfeng Ma, and Yong Qi. Double-edged sword: Incentivized verifiable product path query for RFID-enabled supply chain. In *Proc. of IEEE ICDCS*, pages 414–424, 2017.

[20] Saiyu Qi, Yuanqing Zheng, Mo Li, Yunhao Liu, and Jinli Qiu. Scalable data access control in RFID-enabled supply chain. In *Proc. of IEEE ICNP*, pages 71–82, 2014.

[21] Chen Qian, Yunhuai Liu, R.H. Ngan, and L.M. Ni. ASAP: Scalable collision arbitration for large RFID systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1277–1288, 2013.

[22] Chen Qian, Hoilun Ngan, Yunhao Liu, and L.M. Ni. Cardinality estimation for large-scale RFID systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1441–1454, 2011.

[23] Aysegul Sarac, Nabil Absi, and Stephane Dauzere-Peres. A literature review on the impact of RFID technologies on supply chain management. *International Journal of Production Economics*, 128(1):77–95, 2010.

[24] Muhammad Shahzad and Alex X. Liu. Every bit counts: Fast and scalable RFID estimation. In *Proc. of ACM MobiCom*, pages 365–376, 2012.

[25] Longfei Shangguan and Kyle Jamieson. The design and implementation of a mobile RFID tag sorting robot. In *Proc. of ACM MobiSys*, pages 31–42, 2016.

[26] Longfei Shangguan, Zheng Yang, Alex X. Liu, Zimu Zhou, and Yunhao Liu. Relative localization of RFID tags using spatial-temporal phase profiling. In *Proc. of USENIX NSDI*, pages 251–263, 2015.

[27] Shaojie Tang, Cheng Wang, Xiangyang Li, and Changjun Jiang. Reader activation scheduling in multi-reader RFID systems: A study of general case. In *Proc. of IEEE IPDPS*, pages 1147–1155, 2011.

[28] ShaoJie Tang, Jing Yuan, Xiang-Yang Li, Guihai Chen, Yunhao Liu, and JiZhong Zhao. RASPberry: A stable reader activation scheduling protocol in multi-reader RFID systems. In *Proc. of IEEE ICNP*, pages 304–313, 2009.

[29] Deepak Vasisht, Guo Zhang, Omid Abari, Hsiao-Ming Lu, Jacob Flanz, and Dina Katabi. In-body backscatter communication and localization. In *Proc. of ACM SIGCOMM*, pages 132–146, 2018.

[30] James Waldrop, Daniel W. Engels, and Sanjay E. Sarma. Colorwave: a MAC for RFID reader networks. In *Proc. of IEEE WCNC*, volume 3, pages 1701–1704, 2003.

[31] Chuyu Wang, Jian Liu, Yingying Chen, Lei Xie, Hong Bo Liu, and Sanclu Lu. RF-Kinect: A wearable RFID-based approach towards 3D body movement tracking. *Proc. of ACM UbiComp*, 2(1), 2018.

[32] Ju Wang, Jie Xiong, Hongbo Jiang, Xiaojiang Chen, and Dingyi Fang. D-watch: Embracing bad multipaths for device-free localization with COTS RFID devices. In *Proc. of ACM CoNEXT*, pages 253–266, 2016.

[33] Lei Xie, Jianqiang Sun, Qingliang Cai, Chuyu Wang, Jie Wu, and Sanglu Lu. Tell me what I see: Recognize RFID tagged objects in augmented reality systems. In *Proc. of ACM UbiComp*, pages 916–927, 2016.

[34] Huatao Xu, Dong Wang, Run Zhao, and Qian Zhang. AdaRF: Adaptive RFID-based indoor localization using deep learning enhanced holography. *Proc. of ACM UbiComp*, 3(3):1–22, 2019.

[35] Peizhi Yan, Salimur Choudhury, and Ruizhong Wei. A distributed graph-based dense RFID readers arrangement algorithm. In *Proc. of IEEE ICC*, pages 1–6, 2019.

[36] Lei Yang, Yekui Chen, Xiang-Yang Li, Chaowei Xiao, Mo Li, and Yunhao Liu. Tagoram: Real-time tracking of mobile RFID tags to high precision using COTS devices. In *Proc. of ACM MobiCom*, pages 237–248, 2014.

[37] Lei Yang, Yong Qi, Jinsong Han, Cheng Wang, and Yunhao Liu. Shelving interference and joint identification in large-scale RFID systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3149–3159, 2015.

[38] Jihong Yu, Wei Gong, Jiangchuan Liu, and Lin Chen. Fast and reliable tag search in large-scale RFID systems: A probabilistic tree-based approach. In *Proc. of IEEE INFOCOM*, pages 1133–1141, 2018.

[39] Jihong Yu, Wei Gong, Jiangchuan Liu, Lin Chen, Fangxin Wang, and Haitian Pang. Practical key tag monitoring in RFID systems. In *Proc. of IEEE/ACM IWQoS*, pages 1–10, 2018.

[40] Shigeng Zhang, Xuan Liu, Jianxin Wang, and Jiannong Cao. Tag size profiling in multiple reader RFID systems. In *Proc. of IEEE INFOCOM*, pages 1–9, 2017.

[41] Yuanqing Zheng and Mo Li. ZOE: Fast cardinality estimation for large-scale RFID systems. In *Proc. of IEEE INFOCOM*, pages 908–916, 2013.

[42] Zongheng Zhou, Himanshu Gupta, Samir R. Das, and Xianjin Zhu. Slotted scheduled tag access in multi-reader rfid systems. In *Proc. of IEEE ICNP*, pages 61–70, 2007.

Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems

Yu Liang[1], *Jinheng Li*[1], *Rachata Ausavarungnirun*[2], *Riwei Pan*[1], *Liang Shi*[3], *Tei-Wei Kuo*[14], *Chun Jason Xue*[1]
[1] *Department of Computer Science, City University of Hong Kong*
[2] *TGGS, King Mongkut's University of Technology North Bangkok*
[3] *School of Computer Science and Technology, East China Normal University*
[4] *Department of Computer Science and Information Engineering, National Taiwan University*

Abstract

While the Linux memory reclaim scheme is designed to deliver high throughput in server workloads, the scheme becomes inefficient on mobile device workloads. Through carefully designed experiments, this paper shows that the current memory reclaim scheme cannot deliver its desired performance due to two key reasons: page re-fault, which occurs when an evicted page is demanded again soon after, and direct reclaim, which occurs when the system needs to free up pages upon request time. Unlike the server workload where the direct reclaim happens infrequently, multiple direct reclaims can happen in many common Android use cases. We provide further analysis that identifies the major sources of the high number of page re-faults and direct reclaims and propose Acclaim, a foreground aware and size-sensitive reclaim scheme. Acclaim consists of two parts: foreground aware eviction (FAE) and lightweight prediction-based reclaim scheme (LWP). FAE is used to relocate free pages from background applications to foreground applications. While LWP dynamically tunes the size and the amount of background reclaims according to the predicted allocation workloads. Experimental results show Acclaim can significantly reduce the number of page re-faults and direct reclaims with low overheads and delivers better user experiences for mobile devices.

1 Introduction

With many optimizations to Linux's memory reclaim scheme, the existing Linux memory reclaim scheme can efficiently manage pages inside the main memory in desktops and servers [6, 11, 14, 17]. Android mobile devices, which have seen remarkable growth, inherit the same Linux kernel designed for desktops and servers. As a result, these mobile devices utilize the same memory reclaim scheme inherited from Linux desktop and server distributions.

In this work, we show that the nature of mobile devices workloads is fundamentally different from those of desktop and server workloads. Hence, policies designed to improve the efficiency of the memory reclaim scheme in desktops and servers fail to deliver similar efficiency on mobile devices. Specifically, we observe that a slow page reclaim procedure and severe page thrashing can severely degrade the performance of Android applications. One of the key reason behind this performance degradation is page re-fault, which is a page fault that happens on a previously evicted page. This page re-fault can become a bottleneck and lower workloads' read performance because the system now needs to read the page from the storage instead of the much faster main memory, leading to 100x or more increase in page read latency. Another key reason behind performance degradation is direct reclaim. Direct reclaim can negatively impact performance because any page allocations *must wait* for the direct reclaim process to finish. During a direct reclaim operation, many dirty pages may need to be flushed and thus greatly prolong the allocation procedure. To avoid costly direct reclaim operation, an alternative reclaim scheme using the lightweight background reclaim (***kswapd***), is used by the Linux system. ***Kswapd*** is a kernel thread that is wakened up periodically or by page allocation to reclaim free pages. However, in the current Android kernel, we observe that ***kswapd*** takes too long to free up the necessary number of pages and thus direct reclaim has to be triggered.

Through experiments using popular mobile applications, we show that the current Android memory reclaim scheme does not adapt to the characteristics of Android applications. Experiments show that even when launching one small-size application, page re-faults could happen regularly. Under a set of common use cases, 31% of all the evicted pages are page re-faults. This high ratio of page re-faults to normal page evictions means that page thrashing is very severe in modern Android mobile devices. Aside from the high rate of page re-faults, our experiments also show that the percentage of direct reclaims in all reclaims is 0.8% on average under common use cases. Once direct reclaim happens, up to 1024 dirty pages will be flushed to flash storage, which dramatically extends the latency of the page allocation. Thus, direct reclaim should be avoided.

Prior researches focused on reducing the number of page faults by optimizing page eviction algorithms on mobile devices [32, 40]. These previously-proposed eviction algorithms treat the pages of background and foreground applications with the same priority and choose victim pages according to their access time and the frequency of accesses. The optimized LRU is known as a good eviction algorithm and is applied in Android [2]. To avoid direct reclaim, the Android operating system reserves some free pages by setting watermarks for free memory. This additional free pages can prevent direct reclaim from being triggered if there is a sudden and urgent heavy allocation. However, our experimental results show that the number of direct reclaims is still surprisingly high (could be triggered 96 times in five minutes in one common use case) on Android mobile devices. To reduce the long latency of memory allocation caused by poor insight of mobile OSes, Marvin [28] implements most memory managements in the language runtime, which has more insight into an app's memory usage. However, Marvin misses the opportunities at the OS level, e.g. taking into account the foreground/background states of applications to predict applications' allocation.

In this paper, we observe that under certain user behaviors, page re-fault depends on the amount of available memory, while the direct reclaim depends on both the amount of available memory *and* the latency of background reclaims. Based on these observations, we uncover two main causes that lead to a high rate of page re-faults and direct reclaims on mobile devices. First, background applications are not truly inactive but their reduced activities and unevicted pages still create high memory pressure, penalizing the foreground application in an unfair way. Furthermore, low memory killer (LMK) [1] does not help much. Second, we found that the large-size reclaim, which is suitable for desktop and servers as the large-size reclaim amortizes the long latency of each direct reclaim process, is overly aggressive and coarse-grained for Android because 1) it prolongs the latency of the background reclaim and negatively impacts the user experience and 2) Android workloads typically issue page allocation requests that are much smaller compared to desktop and server requests.

Based on these two main causes, we propose Acclaim, a foreground aware and size-sensitive reclaim scheme. Acclaim consists of two major runtime components: foreground aware eviction (FAE) and a lightweight, prediction-based reclaim scheme (LWP). FAE relocates free pages from background applications to foreground applications; it does so by lowering the priorities of the pages belonging to background applications during page eviction. LWP tunes the sizes and amounts of background reclaims based on its prediction of allocation workloads. Evaluation results show that Acclaim benefits I/O-intensive phases in application execution, notably application launch and application installation, which are known crucial to mobile user experience [8].

The contribution of this paper is listed as follows.

- This work reveals that the current memory reclaim scheme fails to deliver a good page re-fault ratio (of up to 31%) and frequency of direct reclaims (of up to 96 times when using only one foreground application for five minutes) on Android mobile devices.
- We analyze the root causes of the inefficient memory reclaim scheme on mobile devices and propose Acclaim, a foreground aware and size-sensitive reclaim scheme, to improve the performance.
- We conduct a survey to collect the usage information of applications through deploying our monitoring application on fifty-two real mobile devices. We evaluate Acclaim according to our survey. The experimental results on a real mobile device show that the performance improves in most use cases.

2 Background

To analyze the latency bottleneck of Android mobile devices, we first look at how Android read a page of data.

2.1 Android I/O Latency

Android is a Linux-based lightweight operating system designed for mobile devices. Figure 1 shows the architecture of Android I/O stack that including the userspace, the Linux kernel, and the I/O devices.

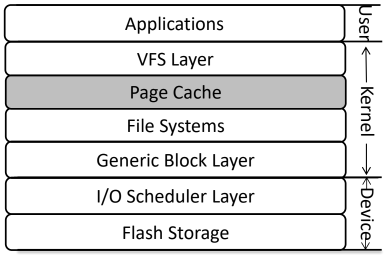


Figure 1: An overview of the Android I/O stack.

We use a read operation as an example to show the latency bottleneck. When an application reads a page in the I/O stack, the application sends a read request to the kernel. The kernel then searches the page cache to see whether the requested page is in the page cache or not. If the requested page is in the page cache, the page cache returns the page to the application. Because the page cache resides in the main memory, the access latency of accessing the page cache takes about one hundred nanoseconds to complete [39]. If the requested page is not in the page cache, a page fault is generated. In this case, the page allocation operation will be triggered to allocate a new page. When the memory is full, the Linux's reclaim scheme is triggered to free pages within the main memory. There are mainly two reclaim schemes: asynchronous background reclaim and synchronous direct reclaim. Background reclaim frees unmapped pages while direct reclaim frees mapped pages or dirty pages, and thus direct reclaim has a heavy cost, especially when writing back dirty pages. After page allocation, the request is delivered to the file system layer, which finds the logical address of the requested page. Then, a read request goes through generic block and I/O scheduler to access the requested page from flash storage through I/O operations. Going through each of these layers contributes to additional microsecond scale latencies to this read request, which can include addressing latency of the file system, queuing latency of the I/O operation, and reading latency of the flash storage. After these operations, the fetched page is finally stored in the main memory and future accesses can be fetched directly from the page cache. Due to these reasons, a page fault can take microseconds to finish, leading to much longer read latency especially when a direct reclaim is triggered.

To quantitatively show the influence of page fault on Android mobile devices, Yu et al. [21] measure the latency of launching Twitter and Facebook applications in three different situations. Figure 2 shows the three scenarios across two setups based on F2FS [19] and EXT4 [23] file systems, which are commonly used in Android. "Cached" refers to the case where most requested pages can be found in the page cache. This case is implemented by re-launching the application right after it is closed, and thus its data is still in memory. "Read" is a case when there are some page faults but there are enough free pages, [1] and thus the reclaim procedure will not be triggered. This case is implemented by launching the application after cleaning the page cache. "Reclaim-first" is the case where there are some page faults and there are not enough free pages, triggering the reclaim procedure. This case is implemented by launching the applications after sequentially launching twenty other applications (to ensure that the page cache is full prior to the launch of Twitter or Facebook.)

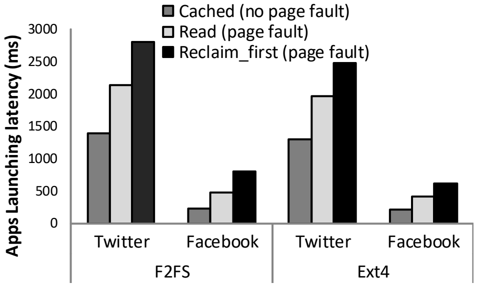


Figure 2: Influence of page fault and reclaim on application launch latency on Android mobile devices.

The results in Figure 2 show that the latency of launching an application is the shortest in the "Cached" case. Compared to the "Cached" case, the "Read" and the "Reclaim_first" cases take longer to launch for both applications. The extended latency is caused by page faults. The launch latency is the longest in "Reclaim_first" case because the reclaim procedure is triggered. Especially, when direct reclaim is triggered, the latency increases significantly.

2.2 Key Factors that Affect Performance

Page Re-fault. Page fault can happen in three scenarios. First, a page fault occurs because physical memory has not yet been allocated for the requested page. This occurs, for example, when the page is read for the first time. Second, a page fault occurs because the application wants to read an already evicted page. We define this case as a page re-fault. Third, a page fault occurs because a process wants to illegally access invalid memory. In this case, the operating system will kill the process. Out of these three cases, the system can be designed to minimize page re-fault because the requested page had been in memory but was evicted by system's page reclaim scheme. Page re-fault can be used to measure the page thrashing and thus evaluate the efficiency of the memory reclaim scheme.

Direct Reclaim. Direct reclaim is a heavy-weight synchronous reclaim scheme that is triggered during the page allocation procedure when there is not enough free space for the system's demands. Once direct reclaim is triggered, Android system needs to pause the allocation process, resulting in additional performance degradation. An alternative solution is to use background reclaim. When the number of free pages is lower than a threshold ($watermark_{low}$), background reclaim threads are woken up to reclaim and free unmapped pages asynchronously. During the background reclaim, the Android system does not pause the allocation process. Hence, background reclaim is lightweight. However, if the background reclaim is unable to reclaim enough free pages in time and there are not enough free pages for the current page allocation, direct reclaim is triggered to reclaim the mapped pages or dirty pages. When the memory is extremely scarce, direct reclaim cannot help and some background applications will be killed by the Android low memory killer (LMK) [1] to reclaim memory. Because the overhead of LMK is larger than direct reclaim [28], LMK cannot be used to replace direct reclaim. Thus, LMK complements direct claim and only handles extreme cases.

3 Analysis of Android Memory Reclaim

In this section, we measure the Android memory reclaim scheme by counting page re-faults and direct reclaims while running popular applications.

3.1 Survey of Application Usage Patterns

We survey the distribution of the numbers of background applications from real phones, then using that numbers to conduct controlled experiments and study launch latencies. We develop a monitoring application [2] and deploy it on the phones of sixty Android users. Out of the 60 users, we verified the data invalidation and selected 52 users (90% 18-35 years old and 10% 35-50 years old) for our analysis. Our

[1] Cache status is checked by the command **dumpsys meminfo**.

[2] https://github.com/MIoTLab/Accliam.

monitoring application collected data on more than twenty mobile device models over a two-month period. During this time, the monitoring application generates an hourly report on other applications' activity, RAM usage, and device information. Our monitoring application runs without root permission. Hence, the application only checks the applications' activity conducted by users but not by systems. With this data, we can estimate the distribution of background applications' usage information as shown in Table 1.

Table 1: Collected data from 52 real phones.

# of phones	# of background applications	Workloads
0	$N < 2$	light
8	$2 \le N < 5$	light
39	$5 \le N < 10$	moderate
5	$N \ge 10$	heavy

Based on the survey, we reproduce different realistic usage scenarios by running several popular Android applications. These applications include Facebook, Twitter, Instagram, WhatsApp, Pinterest, Wish, Chrome, Firefox, Google Earth, Google Map, Uber, Angrybird, CnadyCrush, NewsBreak Youtube, and Spotity. We evaluate both launching and execution of applications with a different number of background applications as shown in Table 2.

Table 2: Application combinations used in experiments. A represents a foreground application and B represents a background application. 3B+A means launching a foreground application when there are three background applications.

Applications	Operations	Memory	Workloads
A	Launch and use an application for 5 minutes	Avail.	Light
3B+A	3 background applications	Avail.	Moderate
8B+A	8 background applications	Full	Moderate
15B+A	15 background applications	Full	Heavy

All our following experiments are performed on a Huawei P9 smartphone with an ARM's Cortex-A72 CPU, 32GB internal memory and 3GB RAM, running Android 7.0 on Linux kernel version 4.1.18. We also conduct experiments on 2.5GB RAM by using **memtester** [34] to occupy memory. There is no external SD card in order to force all the I/O requests to the internal eMMC flash storage (/data partition) of Android. We instrument the Android kernel source code and use the **adb** (Android Debug Bridge) tool [37] to obtain information on memory allocations and the reclaim process of our evaluated smartphone. Our instrumentation framework includes information on the number of re-fault pages, the number of evicted pages, the size of each allocation, the size of each reclaim, the number of direct reclaims, and the number of all reclaim operations. To reproduce the real usage scenarios, after system start, background applications will be launched and wait for a while. And then we start to collect the information while we launch and use the foreground application for five minutes. To avoid bias, each experiment is conducted ten times with the same subset of background applications and the average is shown. In Sections 3.2 and 3.3, we show that page re-fault and direct reclaim happen on Android mobile devices unexpectedly frequently even when a small-size foreground application running.

3.2 Page Re-fault on Mobile Devices

The ratio and the number of page re-faults when launching and running popular applications are shown in Figure 3. We define the page re-fault ratio as the proportion of re-faulted pages on all evicted pages. It can be used to evaluate page thrashing. The results show that the page re-fault ratio could be up to 31% when running popular applications. This means the Android memory reclaim scheme often reclaims pages that will be used soon. Although the ratio of page re-fault depends on users' behaviors, when using only one application in a system with 3GB of memory, page re-faults should not occur because the working set of one application does not exceed 3GB [3]. The existence of page re-faults in Figure 3 indicates that the pre-loaded data and processes' data occupy the memory space causing many page re-faults.

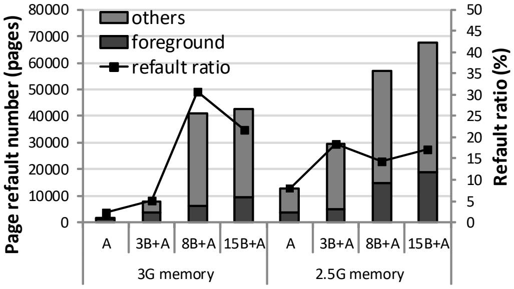


Figure 3: Ratio and number of page re-faults when using one foreground application for five minutes. For each case, there are different number of background applications. "Others" includes background applications and system services.

Moreover, we find that the increase in the number of background applications has a great impact on the number of page re-faults. The ratio of page re-faults is up to 31% when there are eight background applications with 3GB memory. This means almost one-third of the evicted pages will be reused. We further find that a major fraction (37% on average) of page re-faults happens on the foreground app. Because foreground applications directly interact with users, it is important to minimize the number of page re-faults of foreground applications.

3.3 Direct Reclaim on Mobile Devices

Compared to page re-fault, direct reclaim can cause more severe performance degradation and fluctuations because it could flush many dirty pages during a page allocation routine. We show the ratio and number of direct reclaims when running

[3] It is checked by the command **dumpsys meminfo**.

popular applications in Figure 4. We define the direct reclaim ratio as the proportion of the number of direct reclaims on total of reclaims. Even if the direct reclaim ratio is small (up to 2%), it could induce a large latency because page allocations need to wait for the direct reclaim to finish. The latency taken by the direct reclaim can be thousands of times the latency of the background reclaim. Thus, the direct reclaim is supposed to be triggered in memory of emergency cases.

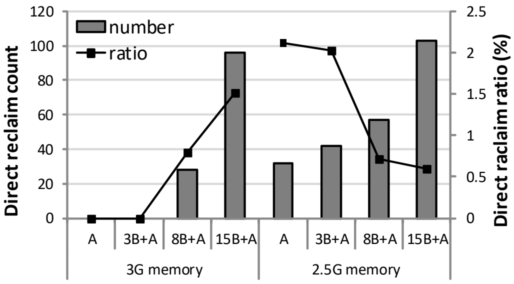


Figure 4: Ratio and number of direct reclaims when using one foreground application for five minutes. For each case, there is different number of background applications.

We find that the increase in the number of background applications and the reduction of physical memory have major effect on the number of direct reclaims. However, the direct reclaim ratio trend is different between on 3GB (default) and on 2.5GB (using **memtester** to occupy memory). When memory is relatively large (3GB), direct reclaim triggers only when there are some background applications and increases as the number of background applications increases. However, when memory is extremely scarce and direct reclaim become ineffective, OS will kill some applications to reclaim memory space. Thus, direct reclaim ratio is decreasing with a larger number of background applications on 2.5 GB of memory. The number of direct reclaims could be up to 96 in five minutes when there are fifteen background applications with the default 3 GB memory. An efficient memory reclaim scheme should minimize the number of direct reclaims.

4 The Cause of Page Re-fault and Direct Reclaim on Mobile Devices

Substantial re-fault rates were also seen on servers, e.g. as high as 14% reported by Google engineers [9]. Compared to servers, mobile devices have vastly different characteristics: much smaller page allocation request size, limited memory, and highly-interactive foreground applications [10, 13]. According to these characteristics, we conduct another set of experiments to analyze the Android memory reclaim scheme to find the main causes of a high number of page re-faults and direct reclaims.

Observation 1: Page re-fault depends on the available memory. Figure 5 shows that the number of page re-fault and evict pages under different available memory. To further eliminate the impact of user behavior, in this experiment, we used different usage scenarios from that in Figure 3. “A” means to launch one foreground app. “A4A” means to launch one foreground application and then launch four background applications, and finally re-launch the foreground application. Relaunching a foreground application is a typical scenario to produce page re-faults.

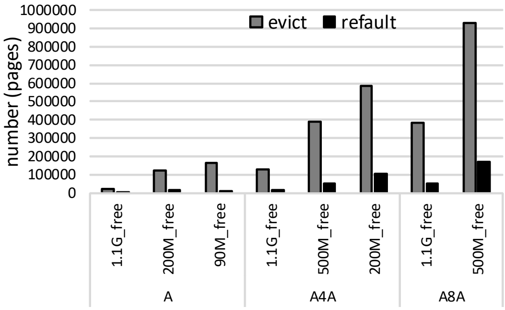


Figure 5: Number of page re-faults and evicted pages under different available memory.

The results show that the number of background applications has a major impact on the number of re-fault and evict pages. Moreover, reducing physical memory can increase number of both re-fault and evict pages. In a word, the number of page re-faults depends on the available memory.

Observation 2: Direct reclaim depends on both available memory and the latency of the background reclaim. The factors which affect the frequency of the direct reclaim can be found in its flow chart which is shown in Figure 6.

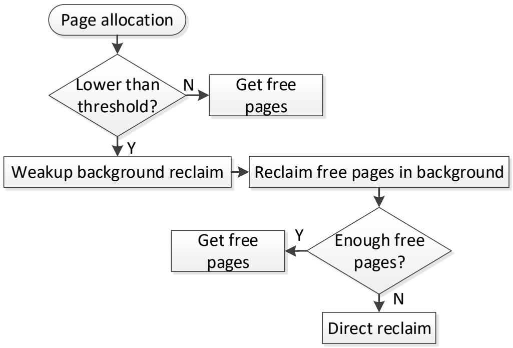


Figure 6: The flow chart of reclaim scheme.

During page allocation, if the number of free pages is lower than a threshold, the background reclaim starts asynchronously. If there are not enough free pages for this allocation or the launched background reclaim does not reclaim enough pages in time, direct reclaim will kick in synchronously. The flow chart shows that direct reclaim depends not only on available memory but also on the latency of the background reclaim. Notably, a larger reclaim size also significantly increases the latency of the background reclaim.

Based on the above two observations and the specific characteristics of mobile devices, we found that there are two additional factors that can increase the number of page re-faults and direct reclaims.

Observation 3: Background applications keep consuming free pages even though they do not have the same impact on user experience compared to the foreground applications. For mobile devices or other highly-interactive systems, foreground applications have significant impact on user experience. However, we found that background applications keep consuming free space under the current memory reclaim scheme due to two main reasons.

First, we find that anonymous pages from background applications thrash pages from the foreground application. In the android system, all the pages are in one of five LRU lists: *Active_anonymous*, *inactive_anonymous*, *active_file*, *inactive_file*, and *unevictable*. The pages in the *unevictable* list will not be evicted. Since anonymous pages contain the heap information associated with a process, these anonymous pages are considered to be more important than file pages to the process. In most cases, the pages in *active_anonymous* list will not be evicted, even if they belong to a background app. Thus, many anonymous pages of background applications stay in memory, while the file pages from the *foreground application* are evicted. These evicted file pages from foreground applications may be accessed again soon, leading to a high number of page re-faults. Moreover, the anonymous pages of background applications occupy free space and thus affect the frequency of the direct reclaim.

Second, we found that background applications are still active even after they are in the background for thirty minutes. The details are shown in Table 3. The results are collected when there are seven user background applications and one foreground application. Notice that system services are treated as applications here.

Table 3: Applications are still active in the background.

Time	Evict apps	Refault apps	Foreground	Background	System
5 mins	53	31	6.2%	34.4%	59.4%
30 mins	52	19	18.3%	33.3%	48.4%

"Evict apps" represents the number of applications that have pages being evicted while "Refault apps" represents the number of applications that have page re-faults. "System" includes system services and private applications that are installed on the mobile device at the factory. The percentages in the Table 3 means the percentage of re-faults. The results show that background applications still request free pages and thus induce page re-faults. Moreover, background applications still consume free space and thus affect direct reclaim frequency.

In summary, the reclaim scheme keeps the pages of background applications in memory and could induce a high number of page re-faults and direct reclaims. For servers, the foreground and background applications usually have the same priority. However, for mobile devices, only the foreground application has a major impact on the user experience.

Observation 4: Large-size reclaim prolongs the latency of the background reclaim. In the buddy system, every memory block has an order, where the order is an integer ranging from 0 to 11. The size of a block of order n is 2^n. The distribution of allocation order when running popular applications is shown in Figure 7. These results are collected from the allocation function *_alloc_pages_nodemask()*. The results show that on the Android mobile device, 99% of allocation orders are 0 (1 page), and more than 99.9% of orders are smaller than 4 (16 pages). This is because the requests on Android mobile devices are mostly in small size. One of the main reasons is that most Android applications use SQLite as the database. SQLite and its temporary files are mostly accessed in 4KB (1 page) units [20, 29].

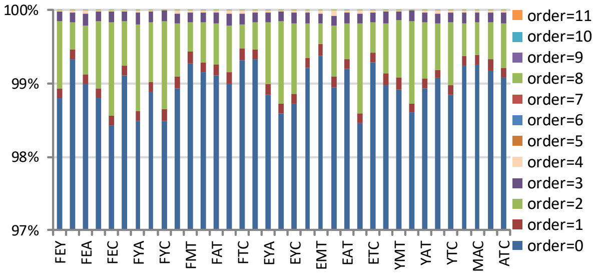


Figure 7: The distribution of allocation orders. The corresponding allocation size equals to 2^{order} [21].

The distribution of reclaim sizes is shown in Figure 8. The results show 80% of reclaim sizes are larger than 32 pages (order=5). Android inherits much of the reclaim scheme from its server counterpart. The latter often reclaims memory as much as possible to fulfill large allocations and avoid expensive direct reclaim or killing processes under memory pressure. As Android applications tend to allocate multiple small blocks of data (shown in Figure 7), the reclaim process becomes overly aggressive and ends up reclaiming excessive pages for each of these small allocations.

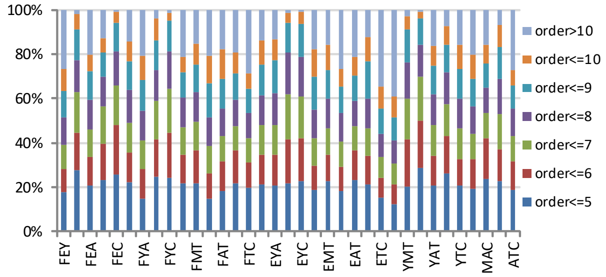


Figure 8: The distribution of reclaim sizes. These results show the reclaims from LRU lists, and they are collected in the functions *shrik_lruvec()* [21].

Large reclaim size prolongs the latency of the background reclaim. According to observation 2, the latency of the background reclaim will affect the frequency of the direct reclaim. Moreover, a large-size reclaim scheme could induce more page re-faults than necessary [21].

Based on our data in Section 3, Android does not efficiently manage its memory. This observation is in-line with multiple technical news: Google Pixel 3 has memory management issues, such as killing background applications [35] and is unable to shuffle between a few applications at a time [33]. Moreover, its memory management issue seemingly gets worse when users use the camera [16]. Hardware vendors tend to address the issue by putting a large-capacity DRAM on devices, which alleviates the problem in a short term but leaves the issue in the future. Moreover, the brute-force solution has many problems, such as cost efficiency, power consumption, the growing trend of application size, etc. and these problems cannot be addressed solely by dropping in more DRAM. Instead, our work aims to improve the efficiency of Android's memory management.

5 Our Solution: Acclaim

With the understanding of the four observations leading to a high number of page re-faults and direct reclaims, we proposed Acclaim, foreground aware and size-sensitive reclaim scheme, which includes two parts. The first part, foreground aware eviction (FAE), is used to solve the problem that background applications keep consuming free pages. FAE takes space from background applications and allocates it to the foreground application. The second part, a lightweight prediction-based reclaim scheme (LWP), is used to reduce the reclaim size of the background reclaim and thus minimize its latency. LWP tunes the size and amount of the background reclaims according to the predicted allocation workloads. In summary, FAE decides from where to reclaim, while LWP decides how much to reclaim.

5.1 Foreground Aware Eviction (FAE)

The memory is always not large enough to eliminate all page re-faults and direct reclaims. Both the number and size of applications increase with memory capacity [38]. Moreover, when the memory capacity increases, mobile device manufacturers often make optimizations, such as locking commonly-used files [3] and pre-loading predicted applications [31] in the memory. All these optimizations consume free memory.

Since the memory size is limited, the total page re-faults can be hardly reduced. The reduction of page re-faults of the foreground application can have a major impact on the user experience of mobile devices or other highly-interactive systems. Thus, we propose to reduce the page re-faults of foreground applications by sacrificing space from background applications. Foreground aware eviction (FAE) is proposed to lower the priority of background pages in LRU lists, causing them to be evicted faster and thus freeing more memory space.

5.1.1 Framework of FAE

The framework of FAE is shown in Figure 9. FAE needs to know whether a page belongs to background applications. Each application has a unique ID (UID). Once an application is installed, its UID is fixed. The UIDs of user applications are added to Page Table Entry (PTE). PTE is only accessible during the page walk process through the page walker. User applications will not be able to access these UID bits as this is handled by OS or hardware. The page's UID is used by FAE during the eviction procedure. Currently, only 8 unused bits of each PTE (the 56th to the 63rd) can be used to store UIDs. Thus, Acclaim only supports 256 unique UIDs at a time, which is an implementation limitation.

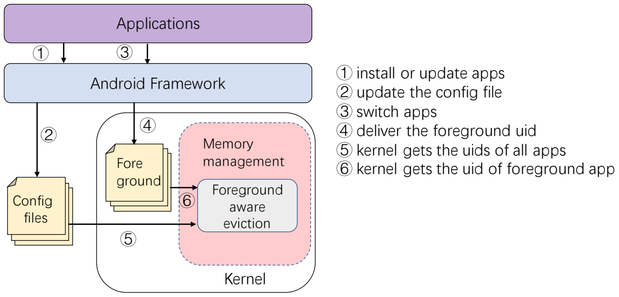


Figure 9: Framework of foreground aware evict scheme.

There is only one foreground application at a moment but there could be several background applications. This list of all background applications can be obtained by subtracting the foreground application from an application list. The main task of FAE is to create a list of background applications and lower their priority compare to the foreground task. To do this, FAE stores the UIDs of applications in the application list in a configuration file. This file will be updated when installing or deleting applications. To identify the current foreground application. FAE notifies the UID of the foreground application to the kernel when users switch applications. Based on the UIDs of applications in the application list and the UID of the foreground application, the system can then lower the priorities of all other applications' pages in LRU lists.

By default, Acclaim deprioritizes all background user applications by assigning them lower priorities in page eviction. To accommodate a small number of applications that keep serving the users in the background, e.g. music or video players, Acclaim can treat them as exceptions without degrading their priorities by excluding them in the application list.

5.1.2 Lower Priority of Background Applications

Initially, we tried to raise the priorities of foreground applications' pages or system's pages. However, this method maintains too many useless pages of foreground applications and the system in memory because of their higher priorities. These useless pages may lead to OS crashes when free memory is used up. Thus, FAE chooses to lower the priority of pages of background applications. Under this scheme, the priority of foreground applications' pages and the system's pages will not be changed. Their useless pages will be evicted from memory and thus free memory will not be used up to crush OS. The details are shown in Figure 10.

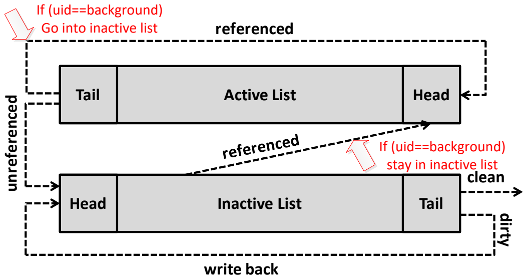


Figure 10: Foreground aware evict scheme of LRU lists.

Page movement between the "active" and the "inactive" LRU lists is driven by memory pressure. Unused pages in the active list go to the inactive list. Pages are taken from the tail of the inactive list to be freed. If the page has the reference bit set, it is moved to the head of the active list and the reference bit is cleared. If the page is dirty, writeback is commenced and the page is moved to the head of the inactive list. FAE lowers the priorities of background pages (See Figure 10) and moves them out of LRU lists quickly. Thus FAE can extract space from background applications for foreground applications and thus to reduce the foreground page re-faults.

Sharing pages have the same priority as their creators. To reduce dynamic-conversion overhead, a sharing page maintains the UID of the application that created it. Thus, the sharing page gets the priority according to the status of its creator. For example, let us assume page A is shared by application D and E. D creates page A first and D is in background while E is in foreground. If page A is not used for a long time, it will be likely to be evicted because its owner is in the background list and thus has low priority. However, if page A is used frequently, A will remains in the memory. Acclaim does not move the application's pages when it is changed from background to foreground and vice versa. Acclaim only checks page's UID and moves a page when it needs to be moved under the default eviction scheme (See Figure 10).

With FAE, the re-launch time of background applications can increase because their pages are out of LRU lists. However, this penalty is much smaller compared to the baseline scenario where these background applications are killed by the Android low memory killer (LMK) [1, 28]. Moreover, the penalty of FAE can be minimized by combining it with application prediction [8, 27, 31]. If the system predicts a background application will be used soon, FAE removes these applications from the background list, and thus does not decrease the priorities of its pages in the LRU lists.

Additionally, FAE is compatible with LMK. For example, FAE can further categorize background applications. Background applications that may be used in the near future can donate some of their memory space with Acclaim while background applications may not be used again can be killed by LMK when memory is getting full.

We expect FAE to benefit impromptu, short interactions (checking and replying instant messages, or switching among applications within a short time span). FAE recognizes them as foreground applications and optimizes them accordingly.

5.2 Lightweight Prediction-Based Reclaim Scheme (LWP)

The original reclaim size of each background reclaim is the maximum number of requested pages until the time of the reclaim. From Section 4, we find that the current reclaim size is too large for the allocation requests of mobile devices. Large-size reclaim induces a high number of page re-faults and direct reclaims. However, there is a trade-off between the reclaim size and performance. If the reclaim size of the background reclaim is too small while the application workloads are heavy, the free pages will be consumed quickly and the heavy-weight direct reclaim will be triggered, lowering performance. On the other hand, if the reclaim size of the background reclaim commands is too large, page re-faults and direct reclaim will happen frequently and, again, degrades overall performance. Thus, we incorporate a workload-prediction based reclaim scheme into our design by incorporating the historical information obtained through a lightweight predictor (See Section 5.2.1). Based on the predicted results, the system can tune the reclaim size of the background reclaim. Another challenge can occur if we reduce the reclaim size according to each workload. In this case, the number of reclaim operations can increase while the amount of reclaimed pages remain unchanged, and thus reducing reclaim size will waste CPU time. To solve this problem, we incorporate the amount of reclaiming pages to dynamically tune the size according to the predicted workloads.

5.2.1 Framework of LWP

LWP consists of two parts, a lightweight predictor and a moderator. Its framework is shown in Figure 11. The lightweight predictor is run during the page allocation procedure. To reduce memory overhead, the sampled allocation requests as inputs are stored in the lightweight predictor. The outputs of the lightweight predictor are the predicted reclaim size and the trend of reclaim amount. The moderator modifies the reclaim size and the amount of the background reclaim according to the predicted reclaim size and the amount trend.

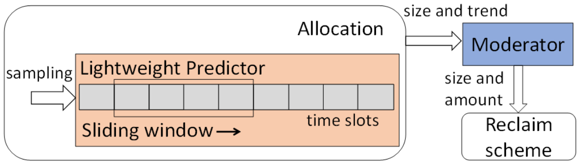


Figure 11: The framework of LWP reclaim scheme.

5.2.2 Lightweight Predictors (LWP)

Using recent information processed by a sliding window to predict the future workloads is commonly applied for prediction [24] [8] [25]. The challenge is how to implement a lightweight predictor. Recent information-based prediction should make sure the correctness of stored information. Page allocation procedure supports concurrency, thus the sliding window needs to guarantee correctness by using locks which will greatly degrade the performance. To predict allocation workloads including size and frequency, the system needs to store the size and time of historical allocation requests. However, storing all historical information precisely will occupy significant space and CPU time and thus degrade the overall performance. To reduce the overhead, the proposed predictor is designed as a lock-free sliding window, and it only stores limited historical information.

Limited historical information. To reduce the stored information, the sliding window is carefully designed in two aspects. First, the sliding window is designed based on time slots to avoid storing the time information. Second, the sliding window stores sampled historical information to reduce the amount of stored information. Each element of the sliding window is the allocation size of a sampling allocation request. For example, when the sampling period equals 10 *ms*, the predictor will pick one of allocation requests that happened in this 10 *ms* and add its allocation size to the sliding window.

Lock-free sliding window. We first analyze the impact of being lock-free on the sliding window. When the window is lock-free, the following three things could happen. (1) Data disorder and data missing could occur; (2) When the predictor samples the historical information, the penalty of being lock-free reduces as there are fewer access to locks; (3) Being lock-free does not affect the system consistency as we only use it to store the memory historical information. We further evaluate the accuracy of the lock-free and sampling predictor. Let sampling = 10 *ms*, sliding = 10 *ms*, and window = 1000 *ms*, to get the sum of the reclaim sizes in a window. To compare three cases, the log of sum value is shown in Figure 12 as the sum in the sampling and lock-free cases are much smaller than that in the lock-free only case and original cases. Three usage behaviors, launching Chrome, launching YouTube, and launching and using five applications, are evaluated. The x-axis is the index of the sliding window. The y-axis is the log of the sum of the reclaim sizes in a window. We use vertical red lines to show the trend changes in the first figure. The results show that the lock-free and sampling case captures the same trend as the original case. (For using the ACFYT case, too much data makes the trend of the sampling case not obvious. It's trend also same as the original case.) Thus, the lightweight predictor can predict the trend of the amount of allocation requests in the next window correctly. In this way, prediction can be achieved with a low overhead in both storage and latency.

5.2.3 LWP-Based Moderator

The moderator is used to tune the reclaim size and amount of the background reclaim according to the predicted results.

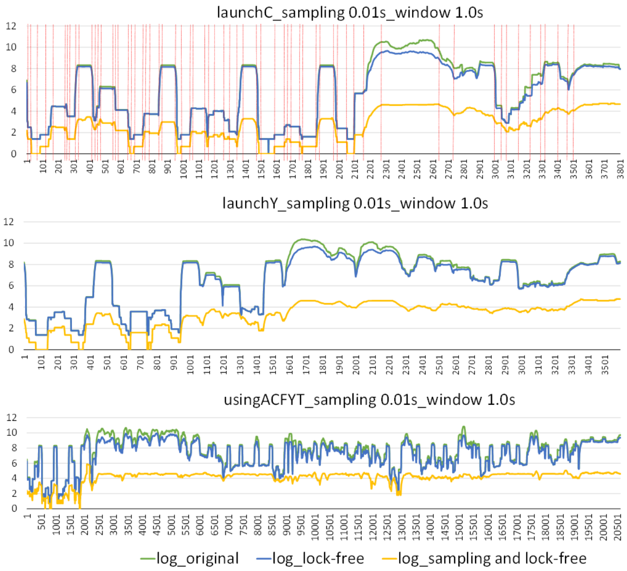


Figure 12: Predicted sum trend of reclaim sizes in the lock-free and sampling sliding window.

The reclaim amount of the background reclaim could be tuned by modifying its stop setting ($watermark_{high}$). The original background reclaim will stop when the number of free pages is above the $watermark_{high}$, which is a fixed value (a proportion of the total number of pages). The original reclaim size of each background reclaim is the maximum number of requested pages until the time of the reclaim. To exploit the trade-off between reclaim size and performance, the LWP base moderator tries to make the background reclaim reclaims just enough free pages just in time. The main idea is to tune the reclaim size of each background reclaim according to the predicted allocation size and tune the $watermark_{high}$ according to the predicted sum trend. "sum trend" is defined as sum/lastsum, where "sum" is the sum of the reclaim sizes in the current window and "lastsum" is the sum of the reclaim sizes in the last window. For the reclaim size, if the trend is larger than a threshold T_1, that means the amount of allocation in the future will be much increased. Thus, the reclaim size should be increased.

- Reclaim size = $P_1 * predicted\ size$ when $trend \geq T_1$
- Reclaim size = $P_2 * predicted\ size$ in other cases, $(P_2 < P_1)$

For the reclaim amount, the moderator tunes the amount of reclaim based on the default value that is set by the mobile device manufacture.

- Reclaim amount = $min(watermark_{high} * (1 + trend), watermark_{high} + T_2)$ when $trend > 1$
- Reclaim amount = $max(watermark_{high} * (1 - trend), watermark_{low})$ in other cases

Reclaim amount is limited. If it is smaller than the default $watermark_{low}$, the performance will be degraded because direct reclaim will be triggered often.

Table 4: Summary of parameters used in LWP.

Symbols	Semantics	Default values
P_1	Amplification factor of reclaim size when I/O is intensive.	4
P_2	Amplification factor of reclaim size when I/O is sparse.	2
T_1	Defines “sudden change” and thus decides the reclaim size.	2
T_2	A threshold to stop background reclaim.	$watermark_{high} - watermark_{low}$

6 Evaluation

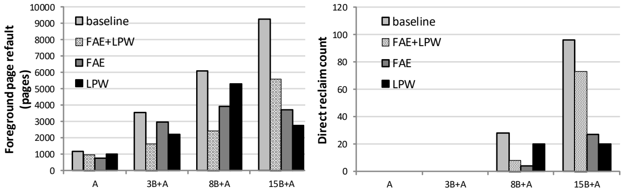
To evaluate Acclaim, we set sampling duration to 10 *ms*, window duration to 100 *ms*, sliding duration to 100 *ms* for sliding window of LWP. Both memory overhead and the predict accuracy of LWP are sensitive to these three parameters.

Moreover, we set the parameters for LWP-based moderator in Table 4. Reclaim size should be larger than the predicted allocation size to avoid memory once heavy workloads are arriving. However, according to the analysis in Section 4, reclaim scheme should not be over aggressive on mobile devices. Thus, we choose small values (4 and 2) as amplification factors (P_1 and P_2) under different workloads.

Furthermore, T_1 determines the sensitivity to the increment in workloads. We configure Acclaim to be sensitive to changes in workloads and responds in time, thus we choose a relatively small value (2). T_2 is the threshold that ensures that the reclaim amount is not too large. Like the default $watermark_{high}$, it is an empirical value. We evaluate Acclaim on three aspects: impact on foreground applications, impact on background applications, and overhead.

6.1 Impact on Foreground Applications

Reduction in page re-fault for foreground applications and direct reclaim of OS. Page re-fault and direct reclaim are closely related to user behaviors. To compare the solution and baseline, we need to choose an application with little change in user behaviors. Thus, a single game, AngryBird, is used as a foreground application for five minutes in this evaluation. The page re-fault and direct reclaim results under the kernel with the original reclaim scheme and the kernel with our solution are shown in Figure 13.



(a) Page re-fault.

(b) Direct reclaim.

Figure 13: The page re-faults in a foreground application (AngryBird) and direct reclaims in the whole OS, showing the benefit of each solution is different in various scenarios.

The results demonstrate the efficacy of Acclaim. For the benchmark (AngryBird), Acclaim reduces page re-faults by 16.3% – 60.2%; it reduces direct reclaims of the whole OS by from 23.9% – 70%. In the experiments, the proposed two techniques play vital, complementary roles. FAE shows higher benefits as the number of background applications increases, as it seizes free pages from background applications to relieve memory pressure. LPW's benefit depends on the accuracy of its prediction, based on which it dynamically tunes the background reclaims. Notably, in case of sudden changes in application workloads, LPW may suffer from accuracy loss and thus underperforms.

Benefit to read/write performance. Reduction in page re-faults and direct reclaims could improve read and write performance. To quantify the impact on read and write operations, we show the read and write performance by using read and write micro benchmarks, [4] and the results are shown in Figure 14. Since most page allocation request sizes on mobile devices are in the size of 4KB [10], we write or read 512MB or 1GB of data in size of 4KB.

This may not be a typical write access pattern, it could happen in some cases. For example, when installing games and applications, more than 1GB of an apk file could be downloaded and written back to flash storage. Moreover, this is a stress test to show Acclaim's benefit under intensive I/O requests. Thus, these evaluation results show the performance impact under intensive I/O requests.

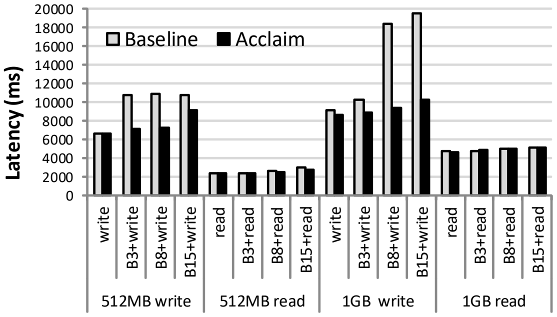


Figure 14: Read and write performance.

The results show that Acclaim improves write performance by up to 49.3% (when writing 1GB of data). This is because page allocation, which Acclaim optimizes, constitutes a significant portion of the delay in writes because of the write-back operations of dirty pages. The read performance is only improved slightly as the latency of page allocation is only a small part of read latency in this set of test cases as no dirty pages are generated and written back during reads.

[4] https://github.com/MIoTLab/Accliam

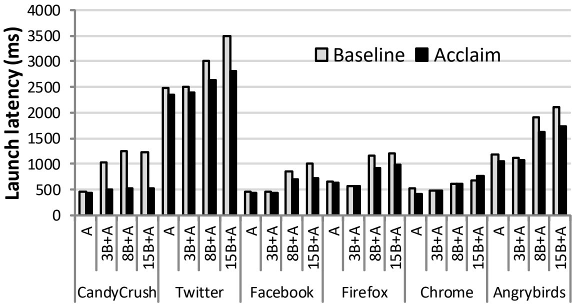


Figure 15: The launch latency of foreground applications.

Benefit to user experience in application launch. To quantify the impact on user experience, the launch time of various foreground applications are evaluated. The evaluation results are shown in Figure 15. The results show that the launch latency improvement varies for different applications. The benefit of Acclaim is more pronounced when an application is launched with multiple memory-hungry applications in background. For example, the launch latencies of CandyCrush and Facebook are reduced by up to 58.8% and 28.8%, respectively. Acclaim can have negative impact when foreground and background applications share common files. For example, the launch latency of Chrome could be prolonged up to 12.3% by Acclaim when there are many background applications. This may because Acclaim evicts common files between background applications and Chrome [3]. However, this penalty can be eliminated by combining with mlocking common files [3]. In summary, Acclaim outperforms the baseline in most test cases. Of all the 24 test cases, it reduces latencies in most of them (20, with median reduction of 19.1% and max reduction of 58.8%) while incurring additional latencies in 4 (with median increase of 3.1% and max increase of 12.3%).

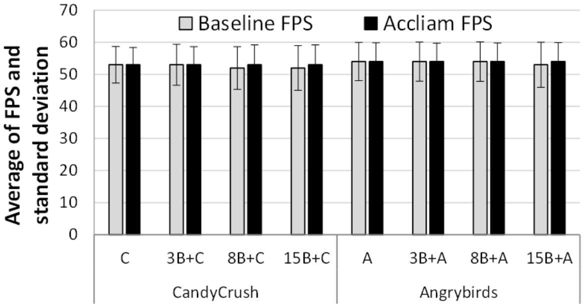


Figure 16: Average FPS and the standard deviation of FPS of foreground applications.

Impact on FPS during active user interactions. Acclaim reduces the launch latency by employing policies on how application uses the memory to cache files. This policy might impact user experience negatively after application launch, as launched applications will have a different amount of data in page cache. Thus, we measure the possible loss in user experience as FPS in KFMARK, a popular gaming benchmark. [12]. The average FPS and the standard deviation of FPS of foreground applications are shown in Figure 16. For the average FPS, the larger the value, the better. While the smaller the value, the better for the standard deviation of FPS. The results in Figure 16 suggest no noticeable impact: the difference between the mean values of the baselines and Acclaim are smaller than their standard deviations.

6.2 Impact on Background Applications

Because Acclaim evicts more pages from background applications, it can negative these applications' re-launch time. We evaluate the re-launch time for the first-launched background application to show the upper bound of the penalty. In this evaluation, we use Facebook as the first launched background application is evaluated when it is launched for one or ten minutes, and the results are shown in Figure 17.

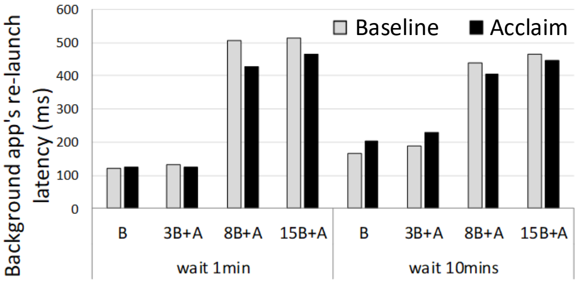


Figure 17: Re-launch time of the first-launched background application (Facebook).

Based on Figure 17, we observe that re-launch penalty only occurs when there are a few background applications after ten minutes. In most cases, the benefit is larger than the penalty. When there are many background applications, Acclaim effectively seizes free pages from background applications, which can be used to aid background applications' re-launch process. Moreover, Acclaim also reduces the number of direct reclaims, benefiting all applications. When the time during which the application is used after it is launched is too short (one minute), Acclaim may not have enough time to move out of the evaluated background application's pages. After system starts, many applications are partially run in the background even if the user does not use them. To this end, their pages will be firstly evicted from LRU lists by Acclaim.

Additionally, notice that the penalty of Acclaim can be eliminated by using it in combination with application prediction [8, 27, 31]. If a background application is predicted as the next used application, Acclaim removes it from the background application list in FAE, and thus the priority of its pages will not be degraded in LRU lists when it is in background. Thus, the penalty will be eliminated.

6.3 Overhead Analysis

Additional memory overhead. FAE adds a uid to PTE, incurring a space overhead of an integer space (4 Bytes). For a device with 3GB of memory, the maximum memory overhead is 3MB. Moreover, FAE needs to store the uids of applications in the application list. If a user installs 100 applications, the total memory cost is only 0.4 KB.

LWP needs to store the sampling historical allocation size (4 Bytes per entry). When the sampling duration is 10 *ms* and the window is 100 *ms*, only 10 values need to be stored. Even if there are 100 values in the window, the LWP only takes up 400B storage overhead. In summary, the total memory overhead is about 3MB (0.1% of memory capacity).

Performance overhead. FAE's performance overhead can be broken down into three parts. First, after system starts, it needs to check the configure file to get the UIDs of the applications. It only happens when the system starts. Second, when the user switches applications, the new foreground UID needs to be delivered to FAE from the framework layer. Third, FAE needs to check if the UID equals to one of the background applications during each page eviction. Only a few comparisons are conducted, thus the performance overhead is negligible.

The performance overhead of LWP includes two parts. First, prediction has a small cost because of the lock-free sliding window. Second, reducing the reclaim size could prolong the wake up time of the background reclaim . However, LWP dynamically tunes the amount of background reclaim according to the allocation workloads to reduce the CPU time consumption. In summary, the performance overhead of Acclaim is trivial.

7 Related Work

Application launch. Existing studies on context-awareness led to the development of application pre-loading algorithms [8, 27, 31]. These algorithms greatly reduce the application launch latency by preparing required resources before they are requested.

Application foreground/background behaviors. Many mobile applications are designed to run in background to enable a model of always-on connectivity and to provide fast response time. This means that once installed and initiated by the user, applications can register themselves with the services provided by the OS framework for background activities, regardless of the user's actual usage of the app. This is true of both iOS and Android OS [4, 7, 26].

Memory management. Many previous works were focusing on the design of the buddy system for managing memory. Burton [6] proposed a generalized buddy system. By using the Fibonacci numbers as block size, Knuth [17] proposed the Fibonacci buddy system. Moreover, this idea was complemented by Hirschberg [15], and was optimized by Hinds [14], Cranston and Thomas [11] to locate buddies in time similar to the binary buddy system. Shen and Peterson [36] proposed the weighted buddy system. Page and Hagins [30] proposed the dual buddy system, an improvement to the weighted buddy system, to reduce the amount of fragmentation to that of the binary buddy system. A buddy system designed for disk-file layout with high storage utilization was proposed by Koch [18]. Brodal et al. [5] improved the memory management for accelerating allocation and deallocation. Marotta et al. [22] proposed a non-blocking buddy system for scalable memory allocation on multi-core machines. Yu et al. [21] show that the existing reclaim scheme is not working well for Android mobile devices. Consequently, this paper proposes a new smart reclaim scheme for Android mobile devices.

Mobile device-specific memory management. Due to mobile OSes have poor insight into application memory usage, the memory allocation may take a long latency, especially under memory pressure. Marvin [28] implements most memory managements in the language runtime, which has more insight into an app's memory usage. They target the same problem at a different layer. By predicting allocation workloads and with foreground and background information, Acclaim improves memory management efficiency at the system level.

8 Conclusion

Existing Linux memory reclaim scheme is designed for servers and PCs. Android inherits Linux kernel and thus the memory reclaim scheme is transplanted to mobile devices. The experimental results show that these algorithms become less effective for the characteristics of applications running on Android mobile devices due to two main reasons. First, background applications has less impact on the user experience than foreground applications. However, they continually consume free pages that increase the frequency of page re-fault and direct reclaim. Second, the large-size reclaim aggravates this problem on mobile devices which involve with almost exclusively small allocation requests. In this work, we propose Acclaim. Acclaim consists of the Foreground aware eviction (FAE), which is designed to relocate free pages from background applications for foreground applications, and the lightweight prediction-based reclaim scheme (LWP), which is used to dynamically tune the size and amount of the background reclaim according to the predicted allocation workloads. Evaluation results show that Acclaim improves the performance in general with a trivial overhead.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Prof. Felix Xiaozhu Lin for their feedbacks and guidance. This paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (11204718) and National Natural Science Foundation of China under Grant No. 61772092.

References

[1] Android open source project. low memory killer. https://source.android.com/devices/tech/perf/lmkd, 2017.

[2] Linux kernel code. lru scheme in the kernel. https://www.kernel.org/, 2019.

[3] Android open source project. mlock commonly-used files. https://source.android.com/devices/tech/debug/jank_jitter, 2020.

[4] AMALFITANO, D., AMATUCCI, N., TRAMONTANA, P., FASOLINO, A., AND MEMON, A. A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software 125* (12 2016).

[5] BRODAL, G. S., DEMAINE, E. D., AND MUNRO, J. I. Fast allocation and deallocation with an improved buddy system. *Acta Informatica 41*, 4 (Mar 2005), 273–291.

[6] BURTON, W. A buddy system variation for disk storage allocation. *Commun. ACM 19*, 7 (July 1976), 416–417.

[7] CHEN, X., JINDAL, A., DING, N., HU, Y. C., GUPTA, M., AND VANNITHAMBY, R. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *MobiCom '15* (2015).

[8] CHU, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys* (June 2012), ACM.

[9] CORBET, J. Proactively reclaiming idle memory. https://lwn.net/Articles/787611/, 2019.

[10] COURVILLE, J., AND CHEN, F. Understanding storage i/o behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)* (May 2016), pp. 1–11.

[11] CRANSTON, B., AND THOMAS, R. A simplified recombination scheme for the fibonacci buddy system. *Commun. ACM 18*, 6 (June 1975), 331–332.

[12] FVIEW. Fps test tool kfmark. https://kfmark.com/, 2017.

[13] GAO, C., SHI, L., XUE, C. J., JI, C., YANG, J., AND ZHANG, Y. Parallel all the time: Plane level parallelism exploration for high performance ssds. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2019), pp. 172–184.

[14] HINDS, J. A. An algorithm for locating adjacent storage blocks in the buddy system. *Commun. ACM 18*, 4 (Apr. 1975), 221–222.

[15] HIRSCHBERG, D. S. A class of dynamic memory allocation algorithms. *Commun. ACM 16*, 10 (Oct. 1973), 615–618.

[16] JANSEN, M. Common google Pixel 3 problems, and how to fix them. https://www.digitaltrends.com/mobile/common-google-pixel-3-xl-problems-and-how-to-fix-them/, 2019.

[17] KNUTH, D. Dynamic storage allocation. *In: The art of computer programming 1*, 435–455.

[18] KOCH, P. D. L. Disk file allocation based on the buddy system. *ACM Trans. Comput. Syst. 5*, 4 (Oct. 1987), 352–370.

[19] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[20] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT)* (2012), ACM, pp. 23–32.

[21] LIANG, Y., LI, Q., AND XUE, C. J. Mismatched memory management of android smartphones. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (2019), USENIX Association.

[22] MAROTTA, R., IANNI, M., SCARSELLI, A., PELLEGRINI, A., AND QUAGLIA, F. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)* (2018), pp. 164–165.

[23] MATHUR, A., CAO, M., BHATTACHARYA, S., AND DILGER, A. The new ext4 filesystem : current status and future plans. In *In Proceedings of Linux Symposium* (2007), pp. 21–33.

[24] MEI, L., HU, R., CAO, H., LIU, Y., HAN, Z., LI, F., AND LI, J. Realtime mobile bandwidth prediction using lstm neural network. In *Passive and Active Measurement* (Cham, 2019), D. Choffnes and M. Barcellos, Eds., Springer International Publishing, pp. 34–47.

[25] MITTAL, G., YAGNIK, K. B., GARG, M., AND KRISHNAN, N. C. Spotgarbage: smartphone app to detect garbage using deep learning. *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2016).

[26] MUCCINI, H., FRANCESCO, A., AND ESPOSITO, P. Software testing of mobile applications: Challenges and future research directions. *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings* (06 2012).

[27] NATARAJAN, N., SHIN, D., AND S. DHILLON, I. Which app will you use next? collaborative filtering with interactional context. pp. 201–208.

[28] NIEL LEBECK, ARVIND KRISHNAMURTHY, H. M. L., AND ZHANG, I. End the senseless killing: Improving memory management for mobile operating systems. In *USENIX Annual Technical Conference (USENIX ATC '20)* (2020), USENIX Association.

[29] Oh, G., Kim, S., Lee, S.-W., and Moon, B. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1454–1465.

[30] Page, and Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers C-35*, 5 (May 1986), 441–447.

[31] Parate, A., Böhmer, M., Chu, D., Ganesan, D., and Marlin, B. M. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2013), UbiComp '13, ACM, pp. 275–284.

[32] Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2006), CASES '06, ACM, pp. 234–241.

[33] Pelegrin, W. Google Pixel 3 is unable to shuffle between a few apps at a time. https://www.androidauthority.com/google-pixel-3-memory-issues-917255/, 2018.

[34] pyropus technology. Memory test tool memtester. http://pyropus.ca/software/memtester/, 2017.

[35] Schoon, B. Google Pixel 3 kills background apps. https://9to5google.com/2018/10/22/pixel-3-memory-management-issue-background-apps/, 2018.

[36] Shen, K. K., and Peterson, J. L. A weighted buddy method for dynamic storage allocation. *Commun. ACM 17*, 10 (Oct. 1974), 558–562.

[37] Shimp208. Android debug bridge (adb) tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool, 2019.

[38] Sims, G. How much ram does your phone really need in 2019? https://www.androidauthority.com/how-much-ram-do-you-need-in-smartphone-2019-944920/, 2019.

[39] Sisoftware. Memory perfromance. https://www.sisoftware.co.uk/author/cas-admin/page/5/, 2017.

[40] Yoo, Y.-S., Lee, H., Ryu, Y., and Bahn, H. Page replacement algorithms for nand flash memory storages. In *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part I* (Berlin, Heidelberg, 2007), ICCSA'07, Springer-Verlag, pp. 201–212.

SweynTooth: Unleashing Mayhem over Bluetooth Low Energy

Matheus E. Garbelini
SUTD

Chundong Wang[*]
ShanghaiTech University

Sudipta Chattopadhyay
SUTD

Sumei Sun
*Institute for Infocomm Research, A*Star*

Ernest Kurniawan
*Institute for Infocomm Research, A*Star*

Abstract

The Bluetooth Low Energy (BLE) is a promising short-range communication technology for Internet-of-Things (IoT) with reduced energy consumption. Vendors implement BLE protocols in their manufactured devices compliant to Bluetooth Core Specification. Recently, several vulnerabilities were discovered in the BLE protocol implementations of a few specific products via a manual approach. Considering the diversity and usage of BLE devices as well as the complexity of BLE protocols, we have developed a systematic and comprehensive testing framework, which, as an automated and general-purpose approach, can effectively fuzz any BLE protocol implementation. Our framework runs in a central device and tests a BLE device when the latter gets connected to the central as a peripheral. Our framework incorporates a state machine model of the suite of BLE protocols and monitors the peripheral's state through its responses. With the state machine and current state of the central, our framework either sends malformed packets or normal packets at a wrong time, or both, to the peripheral and awaits an expected response. Anomalous behaviours of the peripheral, e.g., a non-compliant response or unresponsiveness, indicate potential vulnerabilities in its BLE protocol implementation. To maximally expose such anomalies for a BLE device, our framework employs an optimization function to direct the fuzzing process. As of today, we have tested 12 devices from eight vendors and four IoT products, with a total of 11 new vulnerabilities discovered and 13 new Common Vulnerability Exposure (CVE) IDs assigned. We call such a bunch of vulnerabilities as SWEYNTOOTH, which highlights the efficacy of our framework.

1 Introduction

The Bluetooth Low Energy (BLE) is one of the key wireless communication technologies behind the massive progress of internet-of-things (IoT). Hence, vulnerabilities in the BLE protocol implementation may lead to concrete and serious aftermath. For instance, through reverse engineering on Broadcom's BLE System-on-Chip (SoC) devices, Mantz et al. [24] performed remote code execution in the device's functions with a malformed over-the-air packet. Similarly, BleedingBit [15], discovered in Texas Instruments BLE SoCs, allows adversaries to install a shellcode, which thereafter permits remote execution and authentication bypass upon receiving specific sequences of manipulated advertisement packets.

The preceding examples indicate that faulty BLE protocol implementations may exist in various IoT devices and potentially bring about chaotic consequences. In this paper, we propose a systematic and automated fuzzing framework that is able to discover vulnerabilities in the BLE protocol implementation of any device. Our framework neither requires access to the source code of an implementation nor changes a single line of code in a device's OS or firmware. In a nutshell, it runs in the user space of a customized BLE dongle (i.e., central) to test a BLE device (i.e., peripheral) during the process of establishing a connection between the two.

The essence of our framework is a fuzzer that systematically subjects the BLE implementation to adversarial conditions. However, it is non-trivial to develop a fuzzer to generate such adversarial conditions. Firstly, we construct a BLE state machine model from the Bluetooth Core Specification [36–38] to make valid BLE packets. This is essential, as a randomly generated, meaningless packet is likely to be rejected by any BLE implementation. Secondly, testing a BLE implementation with valid BLE packets is improbable to reveal flaws, because such compliant cases should have been covered in manufacturing tests [22, 41] as well as in Bluetooth stack certification [39]. Thus, our framework either sends malformed packets based on mutation, or normal packets at a wrong time or inappropriate state, or both, to a BLE peripheral. Through manipulating packets, our framework intends to bring on adverse *corner cases*. Thirdly, the complex structure of BLE packets (cf. Figure 1) and the versatile communication regulations necessitate a comprehensive and directed strategy for generating test cases of packets and

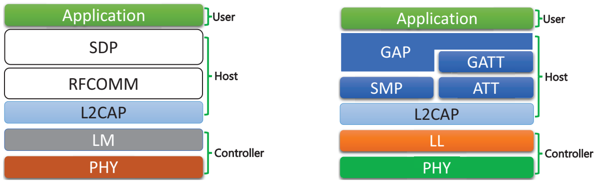
[*] This work was partly done when C. Wang worked at SUTD.

their timings. This aims to drive and stress non-compliant behaviours at the peripheral. To this end, our fuzzer mutates fields of a layer in the BLE stack and employs a particle swarm optimization (PSO) to heuristically refine the mutation probability distribution at both dimensions of each protocol's layers and each layer's fields. Finally, our framework validates any response from a peripheral on-the-fly according to a set of expected packets in each protocol state. This enables it to detect security issues beyond crashes, e.g., security bypass.

Our framework distinguishes itself from existing works [6, 15, 24] in view of being automated and comprehensive. Existing works require manual and tedious efforts, such as reverse engineering and attentive inspection of source code, to discover potential security flaws in the BLE implementation of specific devices [34]. By contrast, our framework is fully automated and embraces the capability to uncover more security issues than a manual approach. Concurrently, although a few scattered approaches have been presented in fuzzing Bluetooth devices [3, 9, 11, 18], they only cover a fraction of the Bluetooth stack. To the best of our knowledge, we compose the first comprehensive approach for BLE fuzzing that is not limited to one or several particular layers, e.g., L2CAP or ATT [3, 11], but fully controls the communication at the Link Layer (LL) as well as the interaction with the Secure Manager Protocol (SMP) for encrypted message exchanging. This, in turn, establishes the efficacy and viability of our framework in fuzzing arbitrary BLE protocol implementations.

The remainder of this paper is organized as follows. In particular, we present the following contributions.

- We present our fuzzing framework to discover implementation flaws for BLE protocols (Section 2).
- We present the optimization process embodied in our fuzzing framework to discover critical security vulnerabilities. We also discuss the systematic process of validating responses from BLE peripheral (Section 3).
- We discuss the implementation specific challenges in our approach and evaluate our fuzzing framework on several commodity BLE SoCs, including SoCs from NXP, Dialog, Texas Instruments, Microchip, ST Microelectronics and Cypress, among others. Our evaluation has revealed 11 unknown security vulnerabilities (nicknamed SWEYNTOOTH) and seven non-compliant behaviours. 13 new common vulnerability exposure (CVE) IDs are assigned and they potentially affect a few hundred types of IoT products. As all the vulnerable SoCs have passed the Bluetooth stack certification, our evaluation also clearly highlights the incompleteness of the certification process (Section 4).
- We evaluate the impact of new vulnerabilities, as discovered by our framework, on four IoT products (Section 4).
- We compare our framework with three other fuzzers and show that our framework is significantly more effective, in terms of finding security vulnerabilities in BLE implementations (Section 4).



(a) Classic (b) BLE

Figure 1: The Stacks of Bluetooth Classic and BLE

After discussing related work (Section 5), we conclude the paper and provide future directions (Section 6).

2 Overview of Our Framework

BLE is the successor of Bluetooth Classic to build a short-range wireless network with reduced energy consumption and improved usage capability. In this section, we first describe the BLE model used in our fuzzing and illustrate the challenges in developing a systematic fuzzing framework for BLE protocols with an example. Then we present an overview of our framework with its main components and workflow.

2.1 The Model of BLE Protocols

We aim to detect *implementation* flaws in BLE protocols defined in the Bluetooth Core Specification [36–38]. Particularly, we study the interactions on Attribute Protocol (ATT), Logical Link Control and Adaptation Protocol (L2CAP), Secure Manager Protocol (SMP), and Link Layer (LL), as shown in Figure 1. L2CAP and ATT are common to both Bluetooth Classic and BLE, while LL and SMP are exclusive to BLE.

Figure 2 illustrates the process of establishing the BLE connection between a central and a peripheral. Our fuzzer works during this process and it is guided by a BLE protocol model we have developed. A simplified representation of the model is presented in Figure 3. Initially, the peripheral periodically broadcasts `advertisements` to nearby devices and the central starts in the `scanning` state. The central scans for such `advertisements` and gets further information from the peripheral such as its name by sending a `scan request` (① in Figure 2). After receiving a `scan response` (② in Figure 2) from the peripheral, the central can choose to start a connection by sending a `connection request` (③ in Figure 2) and proceeds to the `connection` state. On receiving an acknowledgment from the peripheral (④ in Figure 2), the central proceeds to the `initial_setup` state (see Figure 3). As the `connection request` contains connection parameters relevant to the synchronization and communication timing between central and peripheral, after transiting to `initial_setup` state, the central requests information from the peripheral by sending `version request`, `feature request`, `length request` and `MTU length request` (⑤ to ⑧ in Figure 2) with the intention to know the peripheral's

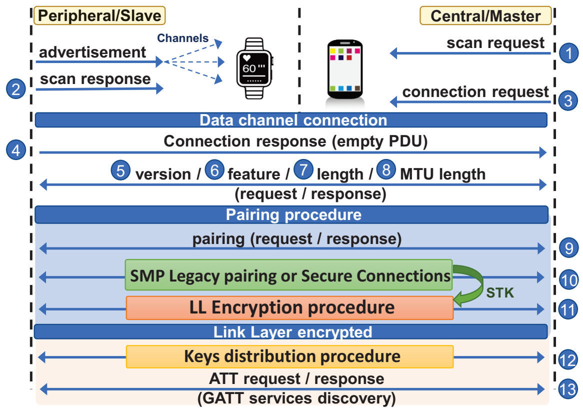


Figure 2: Message exchanges during BLE connection process

supported LL features and capabilities such as the maximum length of the packet it can send or receive. Likewise, the peripheral also gets the central's LL information during the same exchanges. Note that the preceding messages are not necessarily sequentially exchanged, because vendors are free to implement how the peripheral handles such messages. For instance, a peripheral may reply to `version` before `feature`. Similarly, the peripheral may choose to directly read some ATT atributes from the central and go to the `gatt_server` state or skip the state `length` before proceeding. To ensure compatibility with different implementations, we employ several transitions in the state `initial_setup` for the flexible message ordering, as shown at the upper-left of Figure 3.

After the initial setup is done, the central proceeds to the `list_pri_services` state. Here it scans for peripheral's main services via the Generic Attribute Profile (GATT) Service Discovery procedure and stores their attributes in a local array. The central then proceeds to the state `pairing_req` and starts to establish an encrypted communication with the peripheral. The central sends a `pairing request` packet to the peripheral (⑨ in Figure 2), indicating the preferred pairing mode to be used in the next state. If the peripheral accepts the pairing mode proposed by the central, it replies to the central and both proceed to the `smp_pairing` state. As there are two pairing modes for them to choose, i.e., the Legacy pairing or Secure Connection (SC) pairing via SMP exchanges, they go through the pairing procedure from either the `legacy_pairing` or `sc_pairing` state, as shown at the middle-right of Figure 3. Once the pairing procedure is successful, the central derives a `sessionKey` from a Short Term Key (STK) received from `smp_pairing`, transits to the `ll_encryption` state and starts the challenge with the peripheral by sending an `encryption_req` (⑩ in Figure 2). With the peripheral's response, the central sends an encrypted `encryption_res` packet by using the obtained `sessionKey`. If the peripheral is able to correctly authenticate and decrypt the `encryption_res` from the central, it sends another encrypted `encryption_res` to the central, indicating that the connection is successfully encrypted.

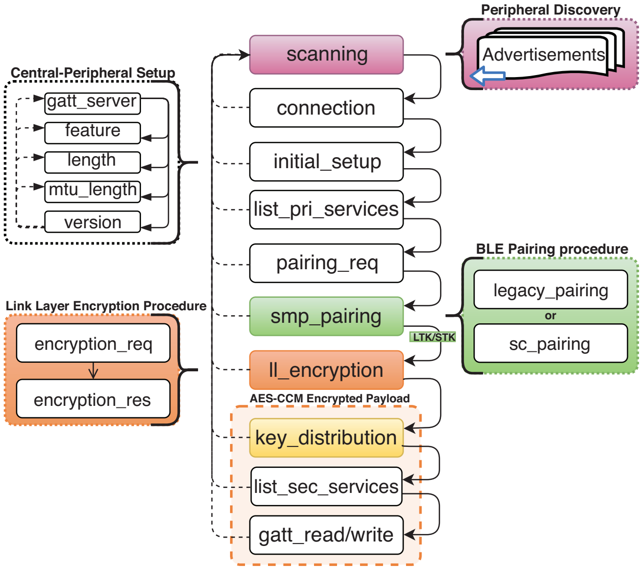


Figure 3: Simplified BLE Protocol Model

If `legacy_pairing` is used, the central and peripheral may optionally go through the `keys distribution procedure` (⑫ in Figure 2) to exchange a long term key (LTK).Otherwise, in `sc_pairing`, the LTK is the STK instead. The LTK can then be used by the central to avoid repeating the pairing process in subsequent connections and directly go to the step ⑪ in Figure 2. In the following stages, the central and peripheral exchange an LTK based on what has been negotiated in `pairing_req` and the central reads more services from the peripheral at the state `list_sec_services`.

After LL connection and pairing, the central discovers all the peripheral's available attributes (i.e., information) by performing the GATT Primary Service Discovery. This consists of sending and receiving a number of `ATT requests` and `ATT responses` (⑬ in Figure 2). so as to fetch predefined ATT attributes. In the next state `gatt_read/write`, we capture the read and write of locally stored ATT attributes at the `list_pri_services` and `list_sec_services` states. This step is to emulate writing malformed ATT attributes via our fuzzing methodology. Thus, the state `gatt_read/write` at the bottom of Figure 3 is not part of the BLE protocol specification. However, it is required to check the behaviour of a peripheral in the presence of malformed ATT attributes.

2.2 Problem Formulation with An Example

In this paper, we consider developing a systematic fuzzing framework that is 1) comprehensive with respect to all BLE stack layers, 2) directed as being with an optimization mechanism to maximally expose anomalies in BLE protocol implementations, and 3) applicable to fuzzing any product embracing BLE SoCs for wireless connectivity. Anomalous behaviours capture non-compliance against the Bluetooth Core Specification. To guarantee the comprehensiveness of cov-

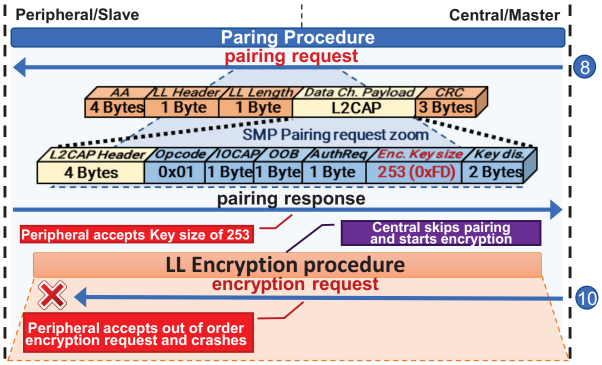


Figure 4: *Key Size Overflow* in Telink SoC (*CVE-2019-19196*)

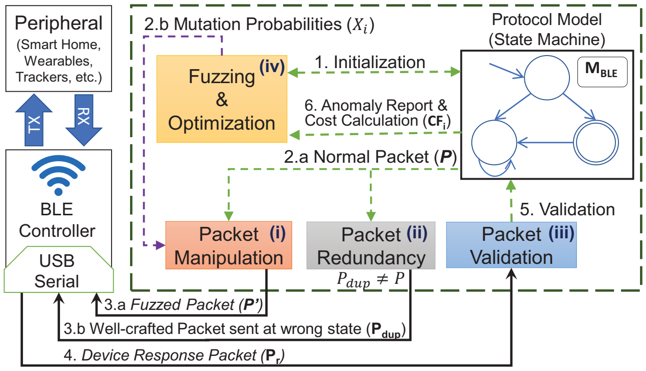

Peripheral
(Smart Home, Wearables, Trackers, etc.)
TX
RX
BLE
Controller
USB
Serial
2.b Mutation Probabilities (X_i)
Protocol Model
(State Machine)
Fuzzing & Optimization
(iv)
1. Initialization
M_{BLE}
6. Anomaly Report & Cost Calculation (CF_i)
2.a Normal Packet (*P*)
5. Validation
Packet Manipulation
(i)
Packet Redundancy
(ii)
Packet Validation
(iii)
$P_{dup} \neq P$
3.a *Fuzzed Packet* (*P'*)
3.b Well-crafted Packet sent at wrong state (P_{dup})
4. *Device Response Packet* (P_r)

Figure 5: An Illustration of Fuzzing Architecture.

ering all protocol layers, we attentively study the Bluetooth Core Specification and incorporate an all-inclusive state machine model as presented in Section 2.1 at the central side. Thus, at the current state of the central, we monitor responses from the peripheral to check whether they are aligned with the Bluetooth Core Specification or not.

Technical Challenges: Section 2.1 indicates that devising a comprehensive state machine model itself is the first challenge due to the complexity of BLE connections. As shown in Figure 1, each of the BLE layers contains multiple fields that might be an exploitable factor. Furthermore, compared to Wi-Fi, BLE allows *move-back* and *move-forward* state transitions if a timeout event occurs and an expected response arrives, respectively. This also introduces the second challenge, i.e., the timing-critical constraints that must be accounted for fuzzing BLE SoCs. Thirdly, an online validation of peripheral responses is non-trivial at the central side. According to the Bluetooth Core Specification, at a given state, the central waits for two types of responses, i.e., normal responses and failure responses. The latter is a valid response, as a well-formed peripheral has the right-of-way to deny any illegal or unaligned request. Such a feature, again, does not exist in Wi-Fi protocols. Consequently, special care is demanded to distinguish expected and anomalous packets in the context of BLE communications. Last but not the least, uncovering vulnerabilities in BLE implementations requires systematically directing the fuzzing framework. In the following, we take an example vulnerability, i.e., *Key Size Overflow* (*CVE-2019-19196*) discovered by our framework, to illustrate how we resolve the aforementioned challenges.

Discovering* Key Size Overflow *Vulnerability: The *Key Size Overflow* vulnerability is caused only if the three following conditions are jointly satisfied: 1) `key_size` field of `SMP pairing request` is fuzzed, 2) the peripheral receives a certain packet in an inappropriate state, and 3) the peripheral may send a connection failure packet depending on the received fuzzed packet. The vulnerability is illustrated in Figure 4.

In brief, as a fuzzer, our framework mutates protocol layers and each layer's fields in a packet sent from the central to the peripheral under test. The mutation is based on probabilities assigned at both dimensions of layers and fields. It refines such probabilities via a cost function with a return value, say, the count of discovered anomalies, to direct the fuzzing process. Our framework identifies an anomaly by validating received responses. It discovers *Key Size Overflow* as follows. Initially, there is no information about the vulnerabilities. Therefore, the mutation probabilities are randomly assigned. Eventually, at the `paring_req` state, the fuzzer sends a `paring_request`, yet with fields other than `key_size` mutated. The peripheral sends a response `SM failure`, which is still deemed to be normal by the online validation of our fuzzer. Next, the fuzzer sends a malformed packet with mutated `key_size`. *Caveat Lector*: the peripheral of Telink Semiconductor unexpectedly replies with a valid `paring_response` for such a fuzzed, invalid request. Our framework legitimately catches this response as an anomaly. As a result, the mutation probability to fuzz the field `key_size` is increased. Thus, more malformed `pairing_request` packets with mutated `key_size` are sent to the peripheral. We note that our fuzzer also sends valid packets, but at an inappropriate state of the client. Eventually, the fuzzer sends an `encryption_request` to the peripheral immediately after the malformed `pairing_request` packets with mutated `key_size`. This crashes the peripheral, as detected due to the lack of any response from it.

To sum up, the `Key Size Overflow` presents an anomaly and a crash for BLE SoCs manufactured by Telink. During the fuzzing process, the scenario to send a malformed `paring_request` (with mutated `key_size`) followed by an `encryption_request` increases. This is because the response to these malformed packets are anomalous and such anomalous responses increase the value of the cost function (i.e., anomaly count). This, in turn, further increases the probability to fuzz `key_size` and indirectly, the likelihood of discovering the scenario causing the vulnerability.

2.3 High Level Workflow

System Architecture: Figure 5 illustrates the architecture of our fuzzer, which is composed of four main modules organized around the BLE model M_{BLE}: (i) the module of packet manipulation that mutates a packet, (ii) the module of packet

Algorithm 1 Main Steps of our fuzzer

```
1:  $i \leftarrow 0$                                  ▷ $i$ captures fuzzing iteration
2:  ▷ generate BLE protocol model (cf. Figure 3)
3:  $M_{BLE} \leftarrow$ Generate_Protocol_Model()
4:  ▷ wait to receive mutation probabilities from PSO
5:  $X_i$ ←-- Particle_Swarm_Opt()
6:  ▷ initialize history of sent packets and redundant packets
7:  $\mathbb{P}_{hist} \leftarrow \emptyset$, $P' \leftarrow \emptyset$, $P_{dup} \leftarrow \emptyset$, $P^h_{dup} \leftarrow \emptyset$, $S_0 \leftarrow \emptyset$
8:  repeat
9:      Set central to be in scanning state
10:     ▷ assign expected layers
11:     For each $S \in M_{BLE}$, assign $\{expected(S), rejection(S)\}$
12:     repeat
13:         Wait for peripheral's packet
14:         Let the central receives packet $P_r$ from the peripheral
15:         ▷ monitor states and checks anomalies
16:         $(\theta_{anom}, P_r) \leftarrow$ Run_Validation$\left(S, P', P^h_{dup}, P_r\right)$
17:         $S_0 \leftarrow S$; $S \leftarrow$ Get_Current_State($M_{BLE}$, $P_r$)
18:         ▷ exit the iteration on anomalies and no transition
19:         if $\theta_{anom}$ is false or $S_0 = S$ then
20:             goto line 37
21:         end if
22:         ▷ generate a valid packet from the model
23:         $P \leftarrow$ Get_Packet_from_Model($M_{BLE}$, $S$)
24:         ▷ generate fuzzed packets from $P$ via mutation
25:         $P' \leftarrow$ Mutate_Packet($P, X_i$)
26:         Send fuzzed packets $P'$ to the peripheral
27:         $P^h_{dup} \leftarrow P_{dup}$
28:         Choose a packet $P_{dup} \in \mathbb{P}_{hist} \cup \{\emptyset\}$ s.t. $P_{dup} \neq P$
29:         Send redundant packet $P_{dup}$ to the peripheral
30:         ▷ switch expected layers after fuzzing
31:         if $P' \neq P$ then
32:             $expected(S) \leftarrow rejection(S)$
33:         end if
34:         $\mathbb{P}_{hist} \leftarrow \mathbb{P}_{hist} \cup \{P\}$
35:     until central does not reach the scanning state
36:     ▷ measure cost function value for $X_i$
37:     $CF_i \leftarrow$ Measure_Cost_Function($X_i$)
38:     ▷ send cost function value to PSO
39:     Particle_Swarm_Opt() ←-- $CF_i$
40:     ▷ wait to receive new mutation probabilities from PSO
41:     $X_{i+1}$ ←-- Particle_Swarm_Opt()
42:     $i \leftarrow i + 1$
43: until timeout
```

redundancy that sends arbitrary packets of M_{BLE} to the peripheral at unaligned states (i.e., out of order) with the intention to trigger anomalies on the peripheral's protocol state machine, (iii) the module of packet validation that is responsible for checking the responses from the peripheral and detecting anomalies based on the current state of M_{BLE}, and (iv) the module of fuzzing & optimization that can direct the mutation of packets based on a cost function.

As shown by the arrows in Figure 5, the four modules of our fuzzer interact and collaborate with each other to attain the aim of discovering potential vulnerabilities in a peripheral device. Algorithm 1 illustrates the workflow of it.

Initialization: The fuzzer relies on the protocol model M_{BLE} to generate valid packets and a set of mutation probabilities X_i to probabilistically mutate such valid packets. At the initialization stage (Lines 3 to 5 in Algorithm 1), the fuzzer first loads the model M_{BLE} and receives initial mutation probabilities X_i from the optimization module (iv in Figure 5) by calling the Particle_Swarm_Opt function (Line 5). Next, the central is set to the scanning state and proceeds to wait for the peripheral's advertisement (Lines 9, 13 to 14). Once the central receives a packet P_r from the peripheral, the validation module (iii in Figure 5) checks whether P_r is expected or not via the Run_Validation function (Line 16). In short, the validation module decides the correctness of P_r based on a set of expected layers expected(S) or rejection layers rejection(S), which are generated for every state $S \in M_{BLE}$ (Line 11) at startup. The validation is detailed in Section 3.2.

Fuzzing Iteration: If the validation does not detect any anomaly, P_r is fed to trigger the state transition in the model M_{BLE} by calling the Get_Current_State function (Line 17). Get_Current_State strictly follows the protocol model described in Section 2.1 and returns the new state S of M_{BLE}. Then at the state S, our framework generates a valid packet P (Line 23), which serves as an input to the manipulation and redundancy modules (i and ii in Figure 5). Starting with the packet manipulation via the Mutate_Packet function (Line 25), the contents of P are mutated according to the mutation probabilities X_i associated with the state S, resulting in a mutated packet P' (see Section 3 for details of mutation). Due to the probabilistic nature of X_i, the mutation yields either an incorrect packet such that $P' \neq P$ (i.e., malformed) or a mutated packet which doesn't differ from the original packet (i.e., $P' = P$). If a malformed packet P' is sent to the peripheral, the Bluetooth Core Specification allows the peripheral to respond with a packet that rejects P', i.e., one with a layer in the rejection(S). Thus, the fuzzer perceives an anomaly if the response for a malformed P' is other than a legitimate packet with one of its layers in rejection(S). To this end, the expected set of layers (expected(S)) for state S is set to the rejection layers for state S (rejection(S)) (Line 32).

The redundancy module (iii in Figure 5) keeps a history $\mathbb{P}_{hist}$ (initialized as $\emptyset$ at Line 7) of all the packets P generated by the model M_{BLE} (Line 34) and sends a redundant packet $P_{dup} \in \mathbb{P}_{hist}$ to the peripheral at random chance (Lines 28 to 29). The intention of this logic is to send out-of-order packets that may cause crash or anomalous behaviour onto the peripheral. However, using redundancy may trigger some ambiguous behaviour which is not necessarily an anomaly. For example, some BLE packets are not only tied to one single state and responses to them at a different state should not be flagged anomalous by the fuzzer. In Section 3.2, we present how the validation module resolves such challenges and avoids reporting false positives.

The fuzzing iteration finishes in one of three circum-

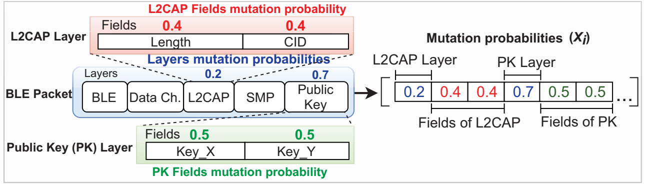


Figure 6: An illustration of our fuzzing. X_i shows the probability values for the packet `public_key` at state S.

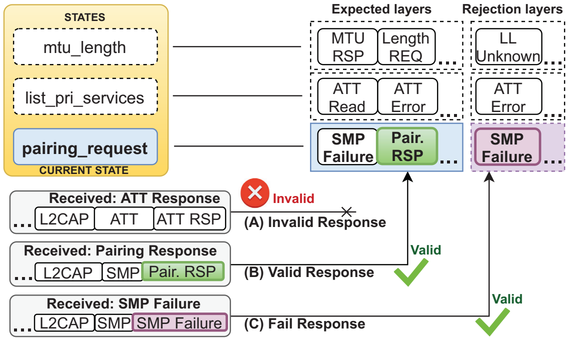


Figure 7: Packet dissection and validation during fuzzing

stances: 1) when the model M_{BLE} reaches the end state `gatt_read/write` (cf. Figure 3) and goes back to `scanning` state, 2) an anomaly is detected (Line 20), or 3) the fuzzer times out due to a crash in the peripheral (Line 13).

Optimization: Once a fuzzing iteration finishes, the mutation probabilities X_i are refined by the optimization module (iv in Figure 5) via particle swarm optimization (Lines 37 to 41). The optimization uses the value of cost function CF_i obtained at the end of every fuzzing iteration (`Measure_Cost_Function`). The rationale of optimization is to guide the mutation probabilities X_i in such a fashion that the value of cost function CF_i is maximized. Specifically, the value of CF_i represents a metric that can direct X_i to fuzz packets that are more likely to optimize CF_i (e.g., the number of anomalies). Moreover, the refined mutation probabilities X_{i+1} are computed iteratively via `Particle_Swarm_Opt` and carried over to the next iteration (Line 41). This approach allows our fuzzer to be directed and facilitates the search for anomalies in the peripheral's protocol implementation.

3 Design of Fuzzer

3.1 Fuzzing and Optimization

The fuzzing effectiveness critically depends on the generation of malformed packets based on mutation. In the following, we discuss how such mutations are performed in detail.

Mutation: On receiving a generated packet from the protocol model, the fuzzing module evaluates it according to the set of mutation probabilities X_i. X_i represents the probabilities to mutate a packet along two dimensions: 1) the `layers`, which correspond to different protocols or packet types of a packet, and 2) the `fields`, each belonging to a layer in the packet. Figure 6 exemplifies the assignment of X_i over the layers and fields of a BLE packet. For instance, consider the Public Key layer to illustrate the use of X_i in generating a packet. The fields of `Key_X` and `Key_Y` can be mutated in an iteration only if the manipulation module randomly hits the layer probability chance (70%). Once a hit happens, the fuzzer needs to decide the set of fields in the layer to be mutated. To this end, the fuzzer iterates over each field within the layer and uses the individual mutation probability (50%) to mutate such fields. We note that all the fields of one layer shares the same mutation probability. This is to reduce the number of parameters during the iterative optimization (cf. Line 39 in Algorithm 1) without losing the efficacy significantly. When the mutation indeed occurs onto a field, the field value is changed via a randomly-chosen `Mutation Operator`.

Mutation Operators: The fuzzing module offers three `Mutation Operators`: 1) **Random bytes** that mutates the value of a packet's fields with random bytes, 2) **Zero filling** that clears the field value to zero, and 3) **Bit setting** that sets the most significant bit of a single-byte field value. The rationale of choosing such operators is to accelerate the search process for an anomaly. In practicality, `Zero filling` and `Bit setting` correlate to setting lower or higher values of a field value to manifest corner cases. These, in turn, are probable to trigger a buffer overflow or underflow in a peripheral's implementation that lacks comprehensive bound checks.

Optimizing Mutation Probabilities: In order to effectively discover anomalies (e.g., crashes or non-compliant behaviours against the Bluetooth Core Specification), our fuzzer employs a cost function to systematically guide the optimization process. The rationale behind such an approach is to measure a cost function value CF_i that informs how well a certain set of mutation probabilities X_i perform with respect to finding new anomalies. Therefore, the goal of the fuzzer is to maximize the discovery of potential anomalies by also maximizing the value of such a cost function. We use the *number of unique anomalies* discovered throughout the fuzzing session as the cost function. This is measured for each individual set of mutation probabilities X_i (cf. Line 37 in Algorithm 1).

The set of mutation probabilities X_i are refined while maximizing the cost function value on each fuzzing iteration by an optimization algorithm (cf. Line 41 in Algorithm 1). For the optimization, we apply the particle swarm optimization (PSO) due to its superior performance in the light of non-linear and stochastic behaviour shown in the protocol model [32]. Moreover, PSO has been successfully applied in a state-of-the-art software fuzzer [23]. The goal of PSO is to optimize the value of a chosen cost function via regulating the *position* of the *swarm of particles* (i.e., the population). In the context of our framework, the *position* is a probability value and each particle within the *swarm of particles* represents a different set of mutation probabilities X_i.

3.2 Packet Validation

The validation module detects responses that deviate from the Bluetooth Core Specification. It emphasizes on the correctness of a response in its *internal packet structure*, i.e., layers of the response, and the *correct reception order*, i.e., the response's arriving state. In particular, given a response packet received at state S, the validation module checks it among `Expected layers` or `Rejection layers` that are dedicated to state S in accordance with the protocol model M_{BLE}.

Validation Exemplified: Figure 7 shows three different cases where a packet from the peripheral arrives in response to a packet sent to the peripheral at state $S =$ `pairing_request`. The packet sent to the peripheral can either be a valid packet P or a mutated packet P'. In **case (A)**, on receiving the `ATT Response` due to a valid P, the validation module flags it as anomaly as none of the layers in the response is found in the `Expected layers` of state S. In **case (B)**, the response packet is deemed to be valid (i.e., `pairing_response`) since its layer is found in the `Expected layers`. On the other hand, after sending a malformed packet to the peripheral, our fuzzer only expects `Rejection layers` (Line 32 in Algorithm 1). In this sense, in **case (C)**, our fuzzer sends a mutated packet P' to the peripheral, and the response with `SMP Failure` is valid as a rejection of P', as `SMP Failure` $\in$ *rejection*(S).

Validation Procedure: More involved cases beyond Figure 7 exist. The validation module must correctly handle responses received due to legitimate, mutated, and/or redundant requests sent at both proper and improper states.

Algorithm 2 illustrates the function `Run_Validation` called in Algorithm 1. It validates if a response P_r is anomalous or not. The response P_r, received at state S, might be due to possible P' and P_{dup} sent in an arbitrary fuzzing iteration (Lines 1 to 5). At start, the validation module prepares the `Expected layers` in ε to be searched for P_r, as P_r might be a response to a non-empty P_{dup} (Line 6 to 10). We first compute the flag Ψ for state S. Ψ holds if the expected layers at S overlap with the expected layers of some other state S' in the protocol model M_{BLE} (Line 7). The flag Ψ does not hold for security-related states such as states involved in SMP pairing and Link Layer encryption, e.g., `smp_pairing` and `ll_encryption`. Specifically, these states (with Ψ false) do not accept any response except those aligned to their respective `Expected layers`. We then check whether a non-empty P_{dup} has been sent at any state M^p_{BLE} (Line 8). The set M^p_{BLE} is a subset of all BLE states (M_{BLE}). Specifically, response to a packet sent at a state $S' \in M^p_{BLE}$ is allowed to be received at any state where Ψ holds (i.e., states other than security-related ones). Thus, given a non-empty P_{dup} sent at a state of M^p_{BLE}, the validation module needs to extend ε if Ψ holds. This is accomplished by joining ε with `Expected layers` of the state P_{dup} belongs to (Lines 8 to 10). With the updated ε, the validation module sets a validity flag based on whether the layers of P_r are expected or not (Lines 11 to 12).

Algorithm 2 `Run_Validation Procedure`

```
 1: Input: Current state $S$ of BLE protocol model (cf. Figure 3)
 2: Input: Packet $P'$ sent from the current state $S$
 3: Input: Packet $P_{dup}$ sent at the immediately preceding state of $S$
 4: Input: Packet $P_r$ sent from BLE peripheral
 5: Output: Absence of anomaly (true or false)
 6: $\varepsilon \leftarrow expected(S)$
 7: $\Psi \leftarrow \exists S' \in M_{BLE} \setminus \{S\}.\ (\varepsilon \cap expected(S')) \neq \emptyset$
 8: if $(P_{dup} \neq \emptyset) \wedge \Psi \wedge (state\_of(P_{dup}) \in M^p_{BLE})$ then
 9:     $\varepsilon \leftarrow \varepsilon \cup expected(state\_of(P_{dup}))$
10: end if
11: ▷ Check if the received packet $P_r$ is valid
12: is_valid $\leftarrow \exists l \in layers\_of(P_r)$ s.t. $l \in \varepsilon$
13: if $(P_{dup} \neq \emptyset) \wedge (state\_of(P_{dup}) \in M^o_{BLE})$ then
14:     $M^p_{BLE} \leftarrow M^p_{BLE} \setminus \{state\_of(P_{dup})\}$
15: end if
16: $M^p_{BLE} \leftarrow (S \in M^o_{BLE})\ ?\ (M^p_{BLE} \setminus \{S\})\ : M^p_{BLE}$
17: ▷ Prevent redundant $P_r$ from transiting the state machine
18: if $(P_{dup} \neq \emptyset)$ and ($P'$ and $P_{dup}$ have the same response) then
19:     Wait for peripheral's response packet $P_r$
20:     Run_Validation($S$, $P'$, $\emptyset$, $P_r$)
21: end if
22: return (is_valid, $P_r$)
```

The validation performs further acts before returning to Algorithm 1. Firstly, in M^p_{BLE} there is a subset, i.e., M^o_{BLE}. The response to the request sent at a state of M^o_{BLE} is allowed to be received in other states, but *only once*. One such state is the `Version` state. A normal peripheral responds to the `version request` only once irrespective of how many `version requests` it receives. Hence, if P_{dup} or P' belongs to some state $S' \in M^o_{BLE}$, then S' is removed from M^p_{BLE}. This ensures that future responses to P_{dup}, which belongs to state S', are classified as anomalies (Lines 13 to 16). Secondly, P_{dup} and P' may have the same response. In this case, we do not trigger a state transition until a response to P' is received (if any before the fuzzer times out). Specifically, after handling the response for P_{dup}, the validation module is recursively called with an empty P_{dup} (Lines 17 to 21). In the end, the anomaly flag and P_r are returned (Line 22).

Crash detection: There are two options to detect a crash or unresponsiveness of the peripheral. The intrusive option is applicable to BLE development boards that expose serial debug ports of their respective SoCs. We can use the debug information to detect a crash. For BLE products without such debug ports, we use a global timer and clear it on every packet response. If no response is received from the peripheral, the timer eventually overflows and a crash is signalized.

3.3 Non-compliant BLE Controller

Manipulation of the Link Layer is essential for fuzzing. However, the Core Specifications [37] undermines Link Layer (LL) manipulation from the host. Firstly, LL packets are heavily timing critical due to BLE frequency-hopping. The host cannot send a packet in a precise time due to the high time variability of the OS scheduler. Secondly, the LL stack runs on a separate and closed source Bluetooth chipset, i.e. the controller. The chipset normally communicates with the host via the Host Controller Interface (HCI) protocol, which does not expose manual control over the LL stack.

To overcome the aforementioned challenges with a *practical* and *low cost* solution, we design a *non-compliant* BLE controller firmware that ignores standardised conventions such as HCI and abstracts away the timing and retransmission requirement between the central and the peripheral. This abstraction simplifies the BLE state machine and allows the host to manipulate all fields of the Link Layer packets.

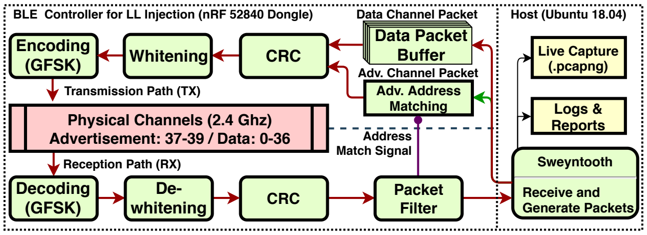


Figure 8: An Illustration of the transmission and reception path of a BLE Packet via the non-compliant BLE controller

Figure 8 details the internals of the non-compliant BLE controller depicted in the fuzzer architecture (cf. Figure 5). The controller reads packets from the host and transmits according to their radio channel type, which is inferred from the *access address* of the packet header. Data channel packets are buffered in the `Data Packet Buffer` and released for transmission after a time period defined by the connection interval. Concurrently, an advertisement packet is only transmitted to the peripheral after the controller receives an advertisement packet from the peripheral first. Upon reception, the `Packet Filter` checks the packet for the peripheral address and upon a match, the advertisement packet stored in the `Adv. Address Matching` is released and transmitted to the peripheral after the inter-frame spacing $\Delta_{IF} = 150us$. Other procedures such as `CRC` calculation, `whitening/dewhitening` and `encoding/decoding` are only necessary to ensure the correct encoding of the packet during over-the-air transmission and as such, do not expose fields for host side fuzzing.

4 Evaluation

Implementation: Our implementation efforts have been mainly spent on two parts: 1) the fuzzer, including the modules of fuzzing, validation and optimization, and 2) the non-compliant BLE controller that enables the over-the-air fuzzing. The fuzzer is written in Python 2.7 and C++ with a total number of 2,836 lines of code (LOC). In brief, our fuzzer extends the `Scapy` v2.4.3 [33] to recognize packet types, parse and validate a response from the peripheral. It also uses the BLESuite library [31] to handle the GATT Service Discovery. As to the fuzzing and optimization, our fuzzer leverages the PyGMO library and its Generational PSO implementation [10] with the optimizer following the common PyGMO structure and the default *pygmo.pso_gen* optimization parameters.

Table 1: Development Platforms used for evaluation

Silicon Vendor	Development Platform	BLE Ver.	Sample Code Name
Cypress (PSoC 6)	CY8CPROTO-63	5.0	Device_Information_Service
Cypress (PSoC 4)	CY5677	4.2	Device_Information_Service
Texas Instruments	LaunchXL-CC2640R2	5.0	project_zero
Texas Instruments	CC2540EMK-USB	4.1	simple_peripheral
Telink	TLSR8258 USB	5.0	8258_ble_sample
STMicroelectronics	NUCLEO-WB55	5.0	BLE_BloodPressure
STMicroelectroncis	STEVAL-IDB008V2	5.0	SlaveSec_A0
NXP	USB-KW41Z	4.2	heart_heart_rate_sensor_bm
Dialog	DA14681DEVKIT	4.2	ble_adv
Dialog	DA14580DEVKIT	4.1	ble_app_peripheral
Microchip	SAMB11 Xplained	4.1	blood_pressure_samb11
Nordic Semi.	nRF51 Dongle	5.0	ble_app_hrs
Nordic Semi.	nRF52840 Dongle	5.0	ble_app_gatts_c

The non-compliant BLE controller is written in C++ (1,096 LOC) within the nRF52840 dongle as the central device. It overcomes the isolation enforced by HCI (cf. Section 3.3).

Evaluation Setup: Table 1 shows the peripheral devices that we have tested. In each of these devices, the CPU is a microcontroller (SoC) that runs an undisclosed BLE stack implementation. IoT products using these devices only have access to interfaces for BLE communications provided by respective manufacturer-provided libraries. As a result, the device's BLE implementation runs alongside the product's main code, and a BLE implementation vulnerability may lead to catastrophic failure and insecurities into the product's functionalities. In other words, once BLE devices are found vulnerable, so are the IoT products relying on them.

We need to install a firmware in each brand new device to enable BLE connectivity. This is accomplished by compiling and programming a sample code provided by the device's corresponding SDK. Once a programmed device advertises itself as BLE peripheral, we can start our fuzzer to test it.

We answer the following research questions (RQs) through the evaluation of our fuzzer.

RQ1: How effective is our fuzzer in terms of generating error-prone inputs?

A summary of testing results is depicted by Table 2. The prefix **V** means a vulnerability while the prefix **A** means some anomalous behaviour that deviates from the legitimate behaviour defined by the Bluetooth Core Specification but is not a vulnerability. Overall, our fuzzer has discovered 11 new vulnerabilities and seven anomalous behaviours over all tested devices. The SoCs of particular vendors, e.g., Texas Instruments, NXP, Cypress and Dialog, have been used in many IoT products for Smart Home, wearables and gadget tracking. These vulnerabilities expose their respective SoCs to crashes, deadlocks or even a complete or partial bypass of pairing procedure. Hence the impact is significant. It's important to emphasize that all vulnerabilities have been automatically discovered by our fuzzer during the packet exchange, except for vulnerabilities classified as `Security Bypass`. After a `Security Bypass` is detected and classified as an anomaly

Table 2: Summary of new vulnerabilities and other anomalies found on the tested platforms. * indicates the case, which is not clear by the Bluetooth Core Specification [36–38]

Vulnerabilities / Inconsistencies		Platform(s)	Model state(s)	Impact Type	Compliance Violated
V1 - Link Layer Length Overflow V2 - Link Layer LLID Deadlock	(CVE-2019-16336, CVE-2019-17519) (CVE-2019-17061, CVE-2019-17060)	CY8CPROTO-063 CY5677 USB-KW41Z	*initial_setup*	Crash Crash, Deadlock	[Vol 1] Part E, Section 2.7
V3 - Silent Buffer Overflow	(CVE-2019-17518)	DA14681 DEVKIT-B	*smp_pairing*	Crash	[Vol 1] Part E, Section 2.7
V4 - Truncated L2CAP Packet	(CVE-2019-17517)	DA14580 DEVKIT-B	*list_pri_services*	Crash	[Vol 1] Part E, Section 2.7
V5 - Unexpected Public Key V6 - DHCheck Skipping	(CVE-2019-17520) (CVE-2020-13593)	LaunchXL-CC2640R2	smp_pairing	Crash Security Bypass	[Vol 1] Part E, Section 2.7 [Vol 3] Part H, Section 2.3.5.6.5
V7 - Invalid connection request	(CVE-2019-19193)	CC2540EMK-USB	*connection*	Deadlock	N.A
V8 - Sequential ATT message	(CVE-2019-19192)	NUCLEO-WB55 STEVAL-IDB008V2	*gatt_read/write*	Crash	[Vol 1] Part E, Section 2.7
V9 - Invalid L2CAP fragment	(CVE-2019-19195)	SAMB11 Xplained	*list_pri_services* *gatt_read_write*	Crash	[Vol 1] Part E, Section 2.7
V10 - Key size overflow V11 - Zero LTK installation	(CVE-2019-19196) (CVE-2019-19194)	TLSR8258 USB	*pairing_req* sc_pairing	Crash Security Bypass	[Vol 3] Part H, Section 3.5.1 [Vol 3] Part H, Section 2.4.4
A1 - Unexpected encryption start response*		SAMB11 Xplained TLSR8258 USB USB-KW41Z	*pairing_request* *smp_pairing*	Non-specified	N.A
A2 - Accept non-zero EDIV and Rand during Secure Connection pairing		LaunchXL-CC2640R2 NUCLEO-WB55 STEVAL-IDB008V2 CY5677	*sc_pairing* *ll_encryption*	Non-Compliance	[Vol 3] Part H, Section 2.4.4.1
A3 - Responds to VERSION_IND more than once A4 - Responds to data channel PDUs during encryption procedure A5 - Sends unknown LL control PDU opcode		TLSR8258 USB	**many** *ll_encryption* *smp_pairing*	Non-Compliance Non-Compliance Non-Compliance	[Vol 6] Part B, Section 5.1.5 [Vol 6] Part B, Section 5.1.3.1 [Vol 6] Part B, Section 2.4.2
A6 - Accepts malformed CONNECT_IND		CC2540EMK-USB	*connection*	Non-Compliance	[Vol 6] Part B, Section 2.3.3.1
A7 - Accepts CONNECT_IND with hopIncrement less than 5		**All tested devices**	*connection*	Non-Compliance	[Vol 6] Part B, Section 2.3.3.1

Table 3: Vulnerabilities and SDK versions of the affected SoCs. * indicates vendors that reported other affected SoCs.

Silicon Vendor	BLE SoC	SDK Ver.	Vuln. / Anomalies
BLE Version 5.0			
Cypress (PSoC 6)	CYBLE-416045	2.10	V1,V2 / A7
Texas Instruments	CC2640R2	2.2.3	V5,V6 / A1,A7
Telink*	TLSR8258	3.4.0	V10,V11 / A3-A5,A7
STMicroelectronics	WB55	1.3.0	V8 / A2,A7
STMicroelectroncis	BlueNRG-2	3.1.0	V8 / A2,A7
Nordic Semi.	nRF51422	11.0.0	A7
Nordic Semi.	nRF52840	15.3.0	A7
BLE Version 4.2			
Cypress (PSoC 4)	CYBL11573	3.60	V1,V2 / A7
NXP	KW41Z	2.2.1	V1,V2 / A1,A7
Dialog*	DA14680	1.0.14.X	V3 / A7
BLE Version 4.1			
Texas Instruments	CC2540	1.5.0	V7 / A6,A7
Dialog*	DA14580	5.0.4	V4 / A7
Microchip	ATSAMB11	6.2	V8 / A2,A7

by our fuzzer, a manual check is required to classify it as a security issue. We note that twelve CVEs have been assigned, but at the time of writing this paper, the details of vulnerabilities V1-V11 were publicly undisclosed for confidentiality. Moreover, we followed responsible disclosures and notified all vendors 90 days in advance for them to provide corresponding patches. At the time of writing, all vendors except STMicroelectronics and Microchip have released their patches. Table 3 highlights the SoCs and the SDK versions where these vulnerabilities were first discovered.

For each anomaly, Table 2 also outlines the specific section of the Bluetooth Core Specification being violated. To summarize, the results signalize that the current status of BLE security demands more attention not only onto the design of protocols, but also onto the implementation phases. Specifically, the two critical security bypass vulnerabilities (V6 and V11) are caused due to the lack of handling corner cases in the Bluetooth Core Specification, causing misinterpretations and implementation flaws. A detailed description of the vulnerabilities is shown in the supplemental material.

Table 4: A Summary of Evaluation Time for Each Device. The connection interval is fixed to 20*ms* for all devices.

Platform	Iterations	Total Time	1st Crash	1st Anomaly	Model Coverage
CY8CPROTO-63	1000	1 h. 06 min.	1 min.	<1 min.	27 (50.0%)
CY5677	1000	2 h. 27 min.	<1 min.	8 min.	29 (53.7%)
USB-KW41Z	1000	1 h. 30 min.	<1 min.	2 min.	24 (44.4%)
DA14681DEVKIT	1000	1 h. 16 min.	10 min.	6 min.	30 (55.5%)
DA14580DEVKIT	1000	2 h. 7 min.	5 min.	1 min.	32 (59.3%)
CC2640R2 Devkit	1000	1 h. 57 min.	4 min.	1 min.	31 (57.40%)
CC2540 Devkit	1000	1 h. 37 min.	2 min.	19 min.	34 (62.96%)
Nucleo-WB55	1000	1 h. 45 min.	<1 min.	2 min.	26 (48.15%)
BlueNRG-2	1000	1 h. 14 min.	<1 min.	9 min.	30 (55.55%)
ATSAMB11	1000	2 h. 39 min.	2 min	10 min.	33 (61.1%)
TLSR8258	1000	1 h. 56 min.	5 min.	<1 min.	36 (66.67%)

RQ2: How efficient is our fuzzer?

When our fuzzer exchanges packets with the peripheral, the efficiency in finding anomalies mainly depends on two factors, i.e., the connection interval and the peripheral's capabilities. While the first factor can be initiated by the central, the peripheral decides whether to accept the value of the connection interval proposed by the central. The connection interval is the time between consecutive messages and thus controls the frequency of messages exchanged between central and peripheral. It is negotiated at the `connection` state. A short connection interval naturally leads to an efficient fuzzing process. During the fuzzing process, the connection interval is fixed to a value that is acceptable to all tested devices. Table 4 shows the overall time taken by our fuzzer to complete 1,000 iterations with a connection interval of 20*ms*. Due to the diverse capabilities of devices, the message-processing time varies significantly even with the same connection interval.

For instance, the CY5677 device is much slower in the pairing procedure, resulting in the longest evaluation time.

The time required to find the first vulnerability in a peripheral's implementation depends on its features. As shown by the rightmost two columns of Table 4, most of the first crash or other anomaly have been discovered within 10 minutes. As a result, our fuzzer is opportune to ascertain a vulnerable implementation of BLE device.

Finally, the last column of Table 4 holds the number of different valid transitions traversed in our BLE state machine (cf. Figure 3) after 1000 iterations. Specifically, the BLE model employs a total of 54 valid transitions. Overall, each peripheral traverses the model differently and does not trigger all possible valid transitions in our BLE model. This is because states *initial_setup*, *list_pri_services* and *list_sec_services* allow multiple transitions and peripheral implementations differ in terms of the exact packet sequence accepted in such states. This results in peripherals missing some transitions employed in the BLE model. As per coverage efficiency, the fuzzer takes more time to fully explore unstable peripherals. This is the case for peripherals impacted by vulnerabilities triggered in states with multiple transitions (V1, V2 and V8). For example, peripherals from Cypress, NXP and STMicroelectronics exhibit a slightly lower coverage value for 1000 iterations.

RQ3: How do the different design choices contribute to the effectiveness of our fuzzer?

To answer this question, we disable two components of our fuzzer to make two variants, respectively. Firstly, we only keep the redundancy module active without packet mutation or optimization. This means packets are sent at a wrong state to the peripheral. Secondly, our fuzzer solely relies on the mutation module without optimization. In this sense, we mutate valid packets from the protocol model M_{BLE} according to a random set of mutation probabilities X_i that is not refined after each iteration. The two variants are referred to as "Redundancy" and "Mutation", respectively.

Figure 9 illustrates the number of anomalies with respect to fuzzing iteration for each relevant BLE SoC. The "Evolution" represents the results achieved by our fuzzer with the optimization, which serves as a reference to compare against the two variants. In all cases, "Evolution" results in finding all anomalies due to the collaborative contributions among all fuzzing components, while the two variants miss some anomalies (cf. Figure 9). This is expected and shows that certain vulnerabilities can only be triggered by either redundancy, mutation or a combination thereof. For example, the vulnerability Key Size Overflow (V10, cf. Section 2) associated with Telink TLSR8258, requires that the mutation and redundancy complement during the fuzzing process to trigger it. That explains the superior effectiveness of "Evolution" in Figure 9(b). Also in Figure 9, "Mutation" cannot achieve as many anomalies as "Redundancy". This is because many anomalies indicated for "Redundancy" are due to the fact that A3 to A5 are triggered upon the peripheral receiving redundant packets in the BLE connection, but not by "Mutation" through sending malformed packets.

Table 5: A Comparison among Testing Tools: *Handcrafted* means tests can be manually configured, whereas a *Test Database* contains a corpus of tests for validation

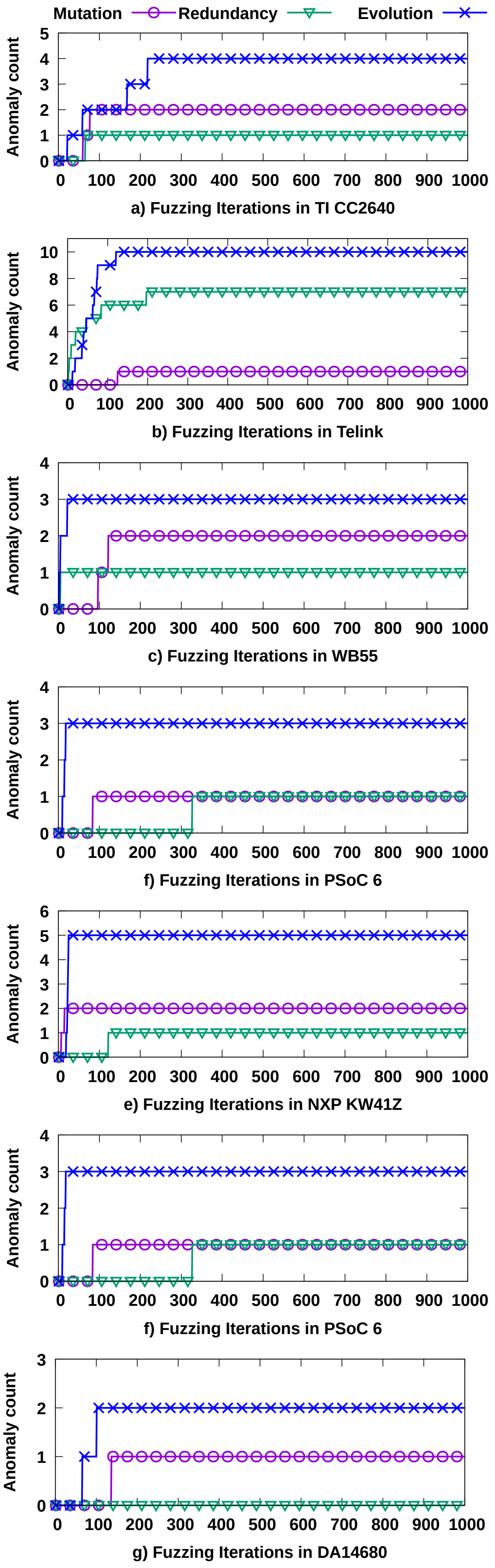
Comparison			Crashes / Anomalies	
Tools	**Supported Layer(s)**	**Fuzzing Strategy**	**WB55, BlueNRG-2**	**Others**
Stack Smasher	L2CAP	Random	0 / 0	0 / 0
BLEFuzz	ATT	Random / Handcrafted	1 / 0	0 / 0
bfuzz (IotCube)	L2CAP	Random / Test database	1 / 0	0 / 0
Our Fuzzer	LL / L2CAP / SMP / ATT	Evolutionary	1 / 2	10 / 7

RQ4: How effective is our fuzzer compared to existing BLE fuzzing tools?

We compare the competitiveness of our fuzzer by evaluating it against publicly available tools, including Stack Smasher, BLEFuzz, and bfuzz that most closely match the objective of our fuzzer. We note that handcrafted efforts were required to apply these tools. Firstly, bfuzz and Stack Smasher demand modifications so that they can send malformed packets through our BLE controller. Secondly, both bfuzz and Stack Smasher were primarily developed for Bluetooth Classic implementations supporting only a few protocols like L2CAP and ATT. Therefore, they also require adjustments for fuzzing L2CAP and ATT layers in BLE implementations. Finally, BLEFuzz is the only tool that supports fuzzing BLE implementations. Table 5 summarizes the comparison between our fuzzer and the three chosen competitors.

For a fair comparison, we run our fuzzer and all the competitors for the same duration (≈ three hours). As shown in Table 5, WB55 and BlueNRG-2 are the only two SoCs for which the competitors discover crashes (third column in Table 5). Specifically, BLEFuzz and bfuzz discovered only V8. For all other SoCs (cf. the "Others" column in Table 5), none of the competitors found either vulnerabilities or other anomalies. In a nutshell, our fuzzer significantly outperforms all competitors, as exemplified in Table 5. The reason is twofold. Firstly, our fuzzer comprehensively models the BLE stack, e.g., it includes modeling and fuzzing SMP and LL protocols, which are not handled by other fuzzers. Secondly, none of the competitors employ an optimization to refine mutation probabilities or send redundant packets. As shown by Figure 9, these features are critical for fuzzing effectiveness.

It is worthwhile to mention that a comparison with the aforementioned tools requires the usage of our non-compliant BLE controller (cf. Section 3.3). This approach is justifiable, *as currently there is no accessible BLE fuzzing alternative with the same level of control and flexibility as provided by our non-compliant BLE controller*. Finally, our comparison did not include traditional fuzzers such as AFL [44] due to their reliance in code coverage. Such a metric is often difficult to obtain in the context of over-the-air-fuzzing, as commercial BLE stacks are undisclosed. Furthermore, traditional fuzzers (e.g. AFL) lack the capability to generate a specific sequence of messages with strict timing constraints. To extend traditional fuzzers with such capabilities requires significant changes to the underlying fuzzing engine. Nevertheless, we



a) Fuzzing Iterations in TI CC2640

b) Fuzzing Iterations in Telink

c) Fuzzing Iterations in WB55

f) Fuzzing Iterations in PSoC 6

e) Fuzzing Iterations in NXP KW41Z

f) Fuzzing Iterations in PSoC 6

g) Fuzzing Iterations in DA14680

Figure 9: Fuzzing effectiveness w.r.t. design components

Table 6: Products verified to be vulnerable

Product	Category	BLE SoC	Vulnerability	Impact
Eve Energy	Smart Home	DA14680	V3	Crash
August Smart Lock	Smart Home			
Fitbit Inspire	Wearables	CY8C68237	V1,V2	Crash
CubiTag	Gadget Tracking	CC2640R2	V5	Deadlock
eGeeTouch TSA Lock	Security	CC2540	V7	Deadlock

envision that even a loose adaptation of traditional fuzzers would yield results similar to Table 5, as anomalies other than crashes cannot be detected out of the box.

Case Studies on IoT Products: The exploitation of SWEYNTOOTH vulnerabilities, as summarized in Table 2, offers dangerous attack vectors against many IoT products. An investigation of certified products on the Bluetooth Listing site [40] reveals that SWEYNTOOTH is likely to affect ≈480 IoT products using the vulnerable SoCs from Table 3. These products are mainly applied in Smart Home, Fitness, Entertainment and Consumer Electronics. To raise awareness of the threats and risks of potentially vulnerable products available on the market, we performed attacks on some representative IoT products that use the affected SoCs and recorded our observations. Some salient features of these products are outlined in Table 6. In Table 6, we also indicate the BLE SoC used by each product and the vulnerabilities discovered in these SoCs by our fuzzer. We choose these products for their prevalence in the relevant application domains, e.g., Smart Home.

To exploit SWEYNTOOTH on an IoT product, we launch an attack code that captures the exact sequence of packet exchanges in the respective SWEYNTOOTH vulnerability. One such example is an attack code for vulnerability `V5` (found in `CC2640R2`) on `CubiTag`. Next, we describe, for each chosen IoT product, the impact of the launched attack code.

When attacking `Fitbit Inspire`, the smartwatch freezes its screen and immediately restarts when the *Link Layer Overflow* (V1) is attempted. By contrast, *LLID Deadlock* stops Fitbit advertisements for several seconds before the smartwatch abruptly restarts. Similarly, when *Silent Buffer Overflow* is exploited on both `Eve Energy` and `August Smart Lock`, users can immediately experience their smart things being restarted (e.g., via a beep sound in the smart lock and switching off the light attached to the `Eve Energy` plug). This is especially crucial for Eve System products, as the company relies almost entirely on the vulnerable DA14680. As for `CubiTag`, the attack exploiting *Public Key Crash* (V5) immediately stops the tracker to advertise and puts it in deadlock. Only a manual restart by opening CubiTag (e.g., via a screwdriver) and re-attaching its battery brings CubiTag back to a working state. Finally, when the *Invalid connection request* (V7) is exploited on `eGeeTouch TSA Lock`, it hangs and the user needs to manually press the *power on* button for further interaction.

5 Related Work

Security is critical for IoT devices [7]. Existing Bluetooth vulnerabilities, such as Blueborne [34], BleedingBit [15] and

KNOB [1], allow unauthorized remote access or remote code execution. They mostly require tedious manual effort (e.g., reverse engineering and inspecting code) and careful inspection of the protocol standard. By contrast, we provide a systematic and automated approach to discover BLE implementation flaws in any BLE device.

Existing works based on static analysis or verification technologies [14, 25, 27, 42] either suffer from false positives or are incapable to generate concrete packet sequences to trigger communication in real devices. An existing test generation approach targeting network protocol implementations [30] require access to the implementation code. Although a recent work `packetdrill` [5] provides a testing framework of the entire TCP/UDP/IP network stack, it lacks support for automated test packet generation. Similarly, Jero et al. [16] devised a technique to search a reduced state-space for suitable attack injection in stateful protocol implementations, but does not employ a comprehensive and directed approach for fuzzing packets. Furthermore, our validation strategy, being employed directly at the central, differs from passive wireless validation [35] that requires a sniffer. Finally, none of the aforementioned works set foot in Bluetooth.

Directed fuzzing is a prevalent software testing strategy [4, 17, 19, 21, 29, 43], yet faces significant challenges in the context of over-the-air fuzzing. Firstly, vulnerabilities in wireless protocol implementation often appear with a sequence of packets being injected even with strict time constraints. Traditional stateless fuzzers such as AFL [44] are mostly suitable for single input leading to vulnerabilities. Secondly, most of the commercial wireless protocol stacks are undisclosed. Thus, it is often not possible to have a greybox (e.g. based on code coverage) or whitebox approach (e.g. based on symbolic execution) for wireless security testing. Thirdly, wireless protocols often exhibit stochastic behaviour, packet drops and packet retransmissions due to the inherent nature of the wireless medium. This introduces additional complexity in security testing, especially in terms of distinguishing normal and abnormal behaviour. Fourthly, wireless protocol stacks often impose isolation between link layer and host layer protocols. A comprehensive security testing should break such isolation to find zero day vulnerabilities. Finally, detecting critical security issues in a wireless implementation, such as security bypass, requires significant changes to the underlying vulnerability detection logic of traditional fuzzers.

Emulation-based fuzzing [13] can obtain coverage information directly from the firmware and is faster than over-the-air fuzzing [26]. Nonetheless, such approaches require extensive reverse engineering of the firmware (if accessible at all) for a substantial number of wireless devices. For example, Frankenstein [20] is an emulation-based fuzzing approach that works with only specific Cypress/Broadcom firmware and demands significant engineering effort to handle other devices.

Previous works in Bluetooth fuzzing [3, 9, 18] support only L2CAP and ATT layers and do not employ test optimization for fuzzing effectiveness. InternalBlue [24] investigates the lower level of Bluetooth implementation and allows BLE packet sniffing and injection. However, InternalBlue can work only after the peripheral is connected and the number of accessible fields in a packet is limited. Our fuzzing framework, by contrast, allows packets injection, fuzzing and sniffing directly from the host and during the BLE connection process.

Our work is orthogonal to several works on network protocol testing [2, 12, 28] that target text structured protocols e.g. `ftp` and `http`, yet they ignore wireless protocols including BLE. A recent work [8] targets the discovery of memory corruptions in IoT devices by fuzzing the mobile app through which the device is accessible. Our work neither intends to fuzz the application layer nor relies on the availability of a mobile app. Moreover, by design of our validation component, our fuzzer can discover security vulnerabilities beyond memory corruptions e.g. security bypass.

In summary, our work is the first comprehensive approach to systematically and automatically fuzz arbitrary BLE protocol implementations. Also, this is accomplished without changing anything in the OS/firmware of tested device.

6 Conclusion

This paper presents a systematic and automated framework for fuzzing arbitrary BLE implementations. This is engineered with the aim to discover implementation behaviours that deviate from Bluetooth Core Specification. The efficacy of this framework is exemplified via the discovery of 11 new security vulnerabilities, named SWEYNTOOTH, across seven BLE SoCs. Moreover, we exploit several SWEYNTOOTH vulnerabilities on popular IoT products used as wearable, smart home products and logistic tracking, among others. This further shows the danger and criticality of SWEYNTOOTH vulnerabilities, potentially affecting a few hundred types of IoT products. Our fuzzer shares the limitation of any framework based on testing. This means, our fuzzer does not *guarantee* the security of a BLE device even if it fails to discover any anomalous behaviour.

SWEYNTOOTH highlights concrete flaws in the BLE stack certification process. We hope that our work provides an opportunity for further research in the area and initiates technologies to harden and secure current and next-generation wireless protocol implementations. For reproducibility and research, the fuzzer source code is available upon request to sweyntooth@gmail.com. All exploits are publicly available in the following URL:

https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks

Acknowledgement: We thank the anonymous reviewers and our shepherd Kevin Butler for their insightful comments. This work is partially supported by Keysight Technologies grant no. RTKS171003.

References

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, Santa Clara, CA, August 2019. USENIX Association.

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security*, pages 343–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] Pierre Betouin. Bluetooth stack smasher version 0.6. http://www.secuobs.com/news/05022006-bluetooth10.shtml, May 2006.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[5] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 213–218, San Jose, CA, 2013. USENIX.

[6] Damien Cauquil. Btlejuice: The Bluetooth smart MITM framework. DEFCON 24, 2016. https://github.com/DigitalSecurity/btlejuice.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, Boston, MA, July 2018. USENIX Association.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA*, February 2018.

[9] Hou-Fu Cheng and Yu-Qing Zhang. Bluetooth OBEX vulnerability discovery technique based on fuzzing. *Computer Engineering*, 34(19):151–153, 2008.

[10] Pagmo development team. Pagmo & Pygmo. https://esa.github.io/pagmo2/, 2019.

[11] Gianluigi Me. Exploiting buffer overflows over Bluetooth: the BluePass tool. In *Second IFIP International Conference on Wireless and Optical Communications Networks, 2005. WOCN 2005.*, pages 66–70, March 2005.

[12] Serge Gorbunov and Arnold Rosenbloom. AutoFuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[13] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, et al. PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[14] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, June 2017.

[15] Armis Inc. Bleedingbit vulnerability. https://armis.com/bleedingbit/, 2018.

[16] Samuel Jero, Hyojeong Lee, and Cristina Nita-Rotaru. Leveraging state information for automated attack discovery in transport protocol implementations. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2015.

[17] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 529–543, 2019.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Poster: Iotcube: an automated analysis platform for finding security vulnerabilities. In *Symposium on Poster presented at Security and Privacy (SP)*. IEEE, 2017.

[19] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 689–701, Santa Clara, CA, July 2017. USENIX Association.

[20] Secure Mobile Networking Lab. Broadcom and Cypress firmware emulation for fuzzing and further full-stack debugging. https://github.com/seemoo-lab/frankenstein, 2020.

[21] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[22] LitePoint. Practical manufacturing testing of bluetooth® wireless devices. https://mcs-testequipment.com/resources/Datasheets_Downloads/Litepoint/Practical-Testing-of-Bluetooth-Devices_WhitePaper.pdf, 2012.

[23] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[24] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 79–90, New York, NY, USA, 2019. ACM.

[25] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[27] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 12, USA, 2004. USENIX Association.

[28] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, April 2017.

[29] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 543–553, New York, NY, USA, 2016. Association for Computing Machinery.

[30] JaeSeung Song ; Cristian Cadar ; Peter Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.

[31] NCC Group Plc. BLESuite libirary. https://github.com/nccgroup/BLESuite, 2019.

[32] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, Jun 2007.

[33] Rohith Raj S, Rohith R, Minal Moharir, and Shobha G. SCAPY- a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, Dec 2018.

[34] Ben Seri and Alon Livne. Exploiting blueborne in Linux-based IoT devices. https://go.armis.com/hubfs/ExploitingBlueBorneLinuxBasedIoTDevices.pdf, 2019. Armis, Inc.

[35] Jinghao Shi, Shuvendu K Lahiri, Ranveer Chandra, and Geoffrey Challen. Wireless protocol validation under uncertainty. *Formal methods in system design*, 53(1):33–53, 2018.

[36] Bluetooth SIG. Bluetooth Core Specification v4.0, June 2010. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[37] Bluetooth SIG. Bluetooth Core Specification v4.2, December 2014. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[38] Bluetooth SIG. Bluetooth Core Specification v5.0, December 2016. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[39] Bluetooth SIG. Bluetooth certification guideline: Qualify your product. https://www.bluetooth.com/develop-with-bluetooth/qualification-listing/, 2019.

[40] Bluetooth SIG. View previously qualified designs and declared products, January 2020. https://launchstudio.bluetooth.com/Listings/Search.

[41] Agilent Technologies. Bluetooth® manufacturing test: A guide to getting started. https://testunlimited.com/pdf/an/5988-5412EN.pdf, 2006. Application Note 1333-4.

[42] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, 2008.

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 724–735. IEEE Press, 2019.

[44] Michal Zalewski. American fuzzy lop. https://github.com/google/AFL, April 2017.

Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds

Yuxin Ren[1], *Guyue Liu*[2], *Vlad Nitu*[3], *Wenyuan Shao*[1], *Riley Kennedy*[1],
Gabriel Parmer[1], *Timothy Wood*[1], *Alain Tchana*[4]

[1] *The George Washington University* [2] *Carnegie Mellon University*
[3] *INSA Lyon France* [4] *ENS Lyon France*

Abstract

5G edge clouds promise a pervasive computational infrastructure a short network hop away, enabling a new breed of smart devices that respond in real-time to their physical surroundings. Unfortunately, today's operating system designs fail to meet the goals of scalable isolation, dense multi-tenancy, and high performance needed for such applications.

In this paper we introduce EdgeOS that emphasizes system-wide isolation as fine-grained as per-client. We propose a novel memory movement accelerator architecture that employs data copying to enforce strong isolation without performance penalties. To support scalable isolation, we introduce a new protection domain implementation that offers lightweight isolation, fast startup and low latency even under high churn. We implement EdgeOS in a microkernel based OS and demonstrate running high scale network middleboxes using the Click software router and endpoint applications such as memcached, a TLS proxy, and neural network inference. We reduce startup latency by 170X compared to Linux processes, and improve latency by three orders of magnitude when running 300 to 1000 edge-cloud memcached instances on one server.

1 Introduction

The Internet of Things foretells the deployment of billions of devices requiring processing close to the data source to avoid excess bandwidth consumption in the network core. Similarly, latency sensitive cyber physical systems desire communication and processing at millisecond scale, preventing the use of standard cloud platforms. Use cases such as these motivate the demand for "edge clouds", tiny data centers that can be deployed as close to users as possible (*e.g.* at an Internet Service Provider (ISP) or a nearby telco central office [55]).

An edge cloud site is expected to serve a large number of clients with high performance. Many edge services such as Network Function Virtualization (NFV) middleboxes that must act as a "bump in the wire" are latency-sensitive and throughput-intensive. However, given the large number of edge cloud sites, each is expected to only have a small number of powerful servers due to space, power, and cost constraints (*e.g.* the HPE EL4000 has 64 cores and AWS Snowball Edge has up to 52 cores). To utilize resources in an efficient, elastic and scalable way, an edge cloud must support dense multi-tenancy—each edge cloud will be *highly resource constrained* compared to a centralized cloud, yet it may need to host many *securely isolated* services for the clients connected to it, and often these clients have a short lifespan (*e.g.* a mobile user), leading to high churn.

Unfortunately, the combination of limited resources, large number of clients, and diverse services of edge clouds pose major challenges for traditional system software designs. To protect clients and services, an edge system needs to provide two types of isolation:

- **Client Isolation**: Multiplexing an edge service among multiple clients exposes them to malicious exploitation that could impact every client (*e.g.* a compromise in the TLS implementation as in Heartbleed). Thus ideally, untrusted clients should not share a protection domain (*e.g.* a process, a container, or a virtual machine).
- **Service Isolation**: An edge server needs to serve multiple services from different tenants. Some services may be vulnerable and tenants may even be malicious. Thus, a service should not share any resources, such as memory, with other untrusted services.

Current systems fail to provide both high performance and strong isolation–particuarly between clients. Recent Network Function Virtualization platforms achieve high throughput with the use of kernel-bypass networking and zero-copy techniques, but they often trade isolation for performance [23, 27, 72]. This works for a single service, but the edge cloud needs to serve multiple services from different tenants. Lightweight virtualization techniques based on unikernels [32] and hypervisor optimizations [47] have been proposed to reduce boot times and density, but don't address providing many isolated clients high throughput. Recent support for HW virtualization, such as SR-IOV capable NICs, reduces virtualization layer costs, but comes at the expense of scalability. It works for a few dozen clients, but can't be used for an edge server that needs to support thousands of clients.

We address these challenges by designing EdgeOS, a new system that achieves the difficult combination of strong isolation, efficient communication, and fast boot times. Our key idea is to dynamically start *a new isolated domain for each client*, and to use *data-copying* to move messages. This idea is based on two intuitions. First, for a large number of short-lived clients, starting a new protection domain can be more efficient and secure than maintaining many long running yet infrequently accessed ones. Second, in contrast to long-standing networking subsystem guidance that dictates that zero-copy is necessary [24, 66] – often at the price of isolation, we observe memory can push data at sufficiently high rates for edge environments such as 5G base stations that have bandwidths in the low 10s of Gb/s [21, 39]. Thus memory copying can provide stronger isolation, without becoming a performance bottleneck as long as it is faster than line-rate.

Based on these insights, EdgeOS contributes:

- A carefully optimized "Memory Movement Accelerator" (MMA) communication and buffer management architecture that enforces isolation with data copying, while retaining high throughput and low latency.
- A "Feather Weight Process" (FWP) that redefines the process abstraction to a minimal memory footprint and set of capabilities needed to support dense deployments of edge computation, and provides strong isolation between each client in multi-tenant environments.
- A control plane with flexible routing and FWP chain caching to support microsecond speed initialization of complex services in high churn environments.

Combined, these features produce a novel architecture that eschews the current trend towards zero-copy I/O in order to provide stronger *per-client* isolation, yet still offers better performance scalability, reduced tail latency, and dramatically better support for high churn edge environments than any system we are aware of.

We extend the Composite μ-kernel [67] to implement the EdgeOS prototype. We target two key categories of edge applications: network functions (e.g., middleboxes from the Click software router [28]) and latency sensitive endpoint services (e.g., HTTPS servers, neural network inference, and the memcached key-value store). These services can be combined to build flexible service chains, while providing stronger isolation and latency guarantees than existing approaches.

Our evaluation illustrates how our isolation and communication abstractions offer dramatically better scale, density, and performance predictability than traditional approaches. We execute 1000s of FWPs per host, instantiate them 170X faster than a Linux process, maintain a `memcached` latency under 1 ms even when running 600 isolated instances on a single host, improve the throughput of HTTPS processing by almost a factor of 2.3, and even CPU-bound neural network inference tail latency improves by almost 50%.

2 Motivation

We first introduce our threat model and isolation properties. Then we discuss performance challenges that edge clouds pose to existing isolation platforms, motivating the need for a redesign of the underlying communication mechanisms and OS primitives.

2.1 Threat Model

There are three types of parties in our model: (1) A system run by the trusted edge cloud operator that provides isolation mechanism and hosts edge services. (2) Edge services deployed by different cloud tenants who supply untrusted code or binaries. (3) Untrusted clients who send requests to the edge services of one or more tenants. The goal of attackers is to compromise security systems, exfiltrate user data, or disrupt edge services. We assume an attacker has capabilities to evade system security mechanisms by exploiting vulnerabilities in the edge service binaries. We consider two general attacker cases: malicious tenants and malicious clients.

Malicious tenant. A tenant could provide malicious or vulnerable services in order to affect the operation of services run by other tenants. After initialization, a tenant's services are trusted only with the permissions given to them by the system for specific resources, such as memory, communication endpoints or system calls. However, a service can make arbitrary use of the permitted resources regardless of whether they are shared by other services.

Malicious client. A client could try to tamper with other clients' traffic by exploiting a vulnerable service. Clients can request any service or send arbitrary packets. After a client successfully attacks a service, we assume it can access any data or resources that are permitted to the controlled service.

EdgeOS seeks to grant resource access permissions to services and enforce isolation among them in order to limit the effects of malicious tenants and clients. In particular, the system wants to maintain tenant-isolation (i.e., a malicious service should not be able to disrupt services from other tenants) and client-isolation (i.e., a malicious client that exploits an instance of a service should not be able to affect other clients). We do not attempt to prevent a malicious tenant's services from affecting its own clients, just as a normal cloud provider does not try to validate client services.

Isolation model. EdgeOS provides a strong form of isolation based on constraining both inter-tenant *and* inter-client (running code for a specific tenant) interference. Tenants provide *chains* of FWPs, each of which executes as a separate, preemptive thread, and protection domain (including page-table-constrained memory). As such, FWPs access disjoint sets of read/write memory, interact only with adjacent FWPs in their chain using message passing, and receive proportional execution time. The lack of shared resources (*e.g.*, no shared memory) and ambient authority (*e.g.*, no shared filesystem namespace) provide strong logical isolation. Preemptive scheduling policies prevent CPU-based resource consumption attacks. EdgeOS ensures that FWPs in a chain cannot be bypassed, and that the output packets cannot be modified by upstream FWPs. As such, FWP chains constitute a high-performance implementation of assured pipelines [7].

We enable a *chain* of FWPs to processes client requests – as opposed to requiring a tenant to provide a *single* FWP– for multiple reasons: (1) FWPs at the start and end of the chain can be required by the system and provide the likes of firewalls and rate-limiting, (2) some applications are naturally implemented in a separate address space, thus using multiple FWPs to provide legacy support, and (3) it allows tenants to more strongly isolate at-risk computations from those that are more important (*e.g.* TLS termination).

EdgeOS's strong isolation between FWPs ensures isolation between tenants. When paired with fast FWP instantiation, it provides per-client isolation. New connections addressed to a tenant's service can (optionally) be served by a *separate* FWP chain, thus lifting the inter-FWP isolation to provide both inter-tenant, and inter-client isolation.

2.2 Existing Isolation Options

In order to support extreme dense per-client isolation, we propose that a protection domain should have the following properties. 1) it is *sealed* [26] so both the binary and configuration cannot be modified after initialization; 2) it has minimal access to system APIs and resources; 3) it cannot share any resource, such as memory, with other untrusted protection domains; 4) once a client's computation is finished, instead of reusing its protection domain – which would allow compromises to impact future executions – it is re-initialized to a safe state.

In contrast, current systems that use process pools or virtualization do *not* provide this inter-instantiation isolation. Existing solutions provide weaker isolation:

- UNIX processes are exposed to large system call interface and TCB in the kernel. Even containers using more security features, such as `cgroups`, `namespaces`, `seccomp-bpf` and `chroot`, still maintain significant state (including signals, file descriptors, memory mappings) that increases attack surfaces.
- Virtualization encapsulates a hardware abstraction along with multiple enclosed processes and system state. Thus it introduces an extra hardware-enforced isolation boundary. Though research has optimized implementations [2,32], the memory overhead, and startup latencies are not sufficient for per-client isolation.
- Language techniques use software-based isolation, but either don't provide temporal isolation, instead executing tenant computation non-preemptively [52] or using heavyweight language runtimes [22].

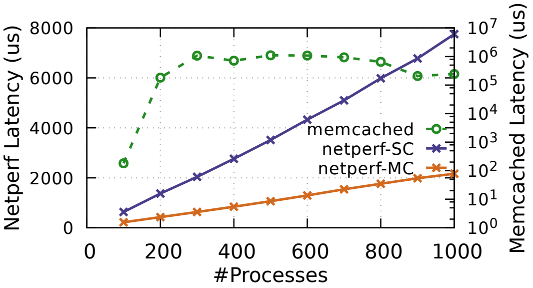


Figure 1: Round-trip latency of N `netperf` or `memcached` instances. Compared with the 1ms round-trip of 5G networks, `netperf` latencies represent a 2x/8x latency increase using one/sixteen cores, while `memcached` exhibits a 1000x latency increase.

2.3 Multi-tenancy and Churn

Given the increasing number of stakeholders that can benefit from edge cloud execution, supporting multi-tenant execution is critical. Network slicing [1, 45, 49] is essential to best utilize edge resources for 5G networking. The challenge [58] is to efficiently share the relatively constrained resources at the edge (often between less than one and low tens of racks [13,14]), while efficiently isolating tenants. Complicating this is the dynamic behavior [43, 44] of these systems which requires adaptation to the environment's inherent churn.

Churn and isolation overheads. Unfortunately, even relatively efficient mechanisms such as containers impose significant overhead when new clients require isolated computation. This is because those mechanisms rely on layers of abstraction and management of a large number of namespaces.

Percentile	Docker	Firecracker	fork()	EdgeOS
50th	521	126	0.26	0.048
90th	574	129	5.8	0.054

The table above depicts the cost in milliseconds of leveraging various isolation facilities; we measure the time to start a minimal service and then fault in 8 pages of memory to show the unpredictability of Linux's Copy on Write (full details in Section 5.2). Using `docker start` can take hundreds of milliseconds due to the cost of initializing namespaces and setting up Docker [10] metadata. Amazon's Firecracker [2, 19] still takes over one hundred milliseconds. Even Linux `fork()`, which has a much lower cost than Docker, exhibits high variance, with the 90^{th} percentile being over 20 times slower than the median. In contrast, our EdgeOS platform improves median start time by 5X compared to Linux, and has minimal variability. Later we show we can improve EdgeOS by another order of magnitude by maintaining a cache of fresh services that can be started near instantaneously.

2.4 Latency and Throughput at Scale

Lightweight isolation mechanisms such as containers facilitate running large numbers of applications (e.g., hundreds of Docker containers per server), but they cannot provide

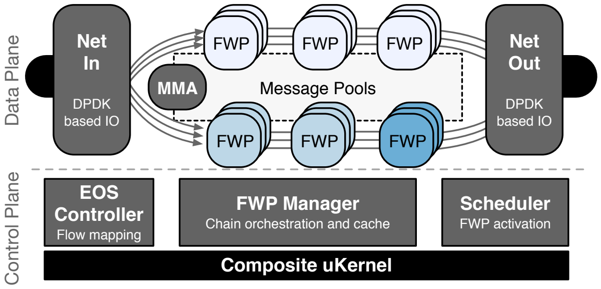


Figure 2: EdgeOS Control and Data Plane Architecture

performance predictability as the scale rises. This leads to the second key challenge in edge infrastructures: predictable performance, particularly latency, at large scale.

Scaling isolation facilities. Unfortunately, current infrastructures suffer poor performance not only under churn, but also at high scale. Both VMs and containers see overheads due to the expense of traversing the host's software switch to determine the appropriate destination to deliver incoming data. Even prevalent and widespread OSes such as Linux suffer from this issue. To evaluate the latency behavior of Linux, we adjust the number of `netperf` servers sharing a single core (`netperf-SC`) or spread across multiple cores (`netperf-MC`), and the number of `memcached` instances spread across multiple cores. A second, well provisioned host transmits traffic to the test server over a 10 Gbps link. Using multiple cores still cannot achieve ideal latency due to poor scalability as shown in Figure 1. Real applications such as memcached are quickly overwhelmed and can only support a hundred or fewer instances (full details in Section 5.6). This illustrates the inability of existing OS isolation mechanisms to provide fine grained performance isolation at high scale. EdgeOS is designed to support isolation with both high scalability and predictability.

3 Design

Figure 2 shows the overall EdgeOS architecture, with trusted components having white lettering. The EdgeOS data plane is composed of Memory Movement Accelerators (MMA) that efficiently and securely copy data between services deployed by tenants as Feather-Weight Processes (FWP), which can be composed into chains to build complex services. The EdgeOS control plane instantiates and schedules these components and routes messages to them.

3.1 Design Principles

Our EdgeOS is designed under the guidance of widely accepted secure system design principles [8, 35, 64].

Avoid shared resources (P1). Every shared resource may open an attack channel [35]. EdgeOS avoids sharing of all types of resources, such as memory, communication endpoints and system services to prevent malicious activities.

Mediated communication (P2). This principle states that communication should be passed via a trusted component, and rules out shared memory based communication. Within EdgeOS, the kernel and MMA mediate all communication initiated from untrusted services.

Least privilege (P3). This well-known principle requires every component to have minimal privileges to limit damage from a system compromise. However, current isolation mechanisms, such as containers or VMs, are usually running on top of monolithic systems, whose kernel or hypervisor has full privilege. EdgeOS applies this principle not only to untrusted components, but also low-level system services, such as MMA and scheduler.

3.2 Memory Movement Accelerators (MMA)

Copy-based communication. Existing high throughput systems [51, 52, 72] often eschew isolation and use shared memory to pass data among isolated services. In contrast, EdgeOS eliminates shared memory between services (**P1**). A key EdgeOS design is that all communication between untrusted services use *data copying* and are mediated by MMA (**P2**).

As long as memory copying is higher bandwidth than the network line-rate, it is a viable form of data movement that provides strong isolation. On our processor, memory throughput is 472 Gb/s, and though networking throughput is ever-increasing in the data-center, it is more limited on the edge. Practically, in §5.1 (Figure 5(c)) we show that the MMA can sustain throughput competitive with a middlebox framework – that avoids copying by sharing packet memory – up to 54 Gb/s. For perspective, 5G cells provide on the order of between 2Gb/s [21] and 20Gb/s [39]. This design is counter to long-standing networking subsystem guidance that dictates that zero-copy is necessary [24, 66] – often at the price of isolation, EdgeOS optimizes the MMA implementation and treats it as a specialized processor (§4.2) to achieve the line-rate. As a result, EdgeOS maintains strong isolation without practically sacrificing performance.

Efficient data copying with the MMA. The MMA acts as a software DMA engine to move message data between services, and runs on one or more *dedicated cores* in order to perform out-of-band data movement. The MMA retrieves messages from an upstream service's message rings, copies them and adds them into a downstream service's message rings, and alerts the scheduler that the destination service needs to be activated to receive it. EdgeOS further separates memory into a *message pool* that is used for communication, and *local memory* for each service's local state. This separation enables memory allocations to be optimized for the purpose and use of the memory (§4.1). MMA only has the access rights to copy into, or from message pools (**P3**).

Network Gateways. In and Out gateways leverage DPDK [11], run on dedicated cores[1] and pull packets into message pools with no kernel interactions. MMA then copies

[1] Note that current edge offerings include between 20-64 cores, and we show (§5) that, in aggregate, EdgeOS is efficient despite specializing cores.

packets into the destination service, thus enabling strong protection among both (untrusted) edge services and (trusted) system services.

3.3 Feather-Weight Processes (FWP)

A Feather-Weight Process (FWP) is a minimal abstraction wrapping only memory and a small set of simple kernel resources. FWP achieves strong isolation by (1) capability-based access control which minimizes access to the rest of the system; (2) library-based services to avoid sharing of sensitive information; (3) FWP caching that re-initializes a FWP's context before serving a new client.

Capability-based resource isolation. In EdgeOS design, access to all resources relies on capability-based access control [9] using kernel-mediated references, removing any ambient authority [37] (**P2, P3**). These resources include local memory, the message pool that is used to receive and send data, and synchronous communication end-points to request operations from system-level services. Capabilities to memory are enforced by hardware page-tables, while other capabilities are protected by the kernel. Each FWP is encapsulated within its own capability space, which restricts the granted resource to only the allowed tenant. No memory is shared between FWPs, instead MMA copies data between FWPs.

Library-based services. Notably absent in EdgeOS are default access to conventional shared OS services. Similar to Unikernels [30, 31, 69], FWPs make use of library-based implementations [18], thus enabling the inclusion of only the application-required services without sharing with other FWPs (**P1**). We have ported a TCP/IP networking stack and a simple in-memory file-system to FWPs. The memory-based file system is used to store transient and configuration data. If global persistent state is required, then network-accessible storage services can be used (similar to a serverless computing model). This decoupling enables a simplified and efficient FWP execution environment to enable high density and line-rate computation.

FWP Caching. A new client should not be allowed to see old context accumulated from previous clients. Current practices either ignore this, such as process pools, or manually terminate and restart the service [3, 5], which repeatedly incurs unnecessary initialization overhead. EdgeOS employs an FWP checkpoint cache, that both avoids reusing possibly compromised state of previous executions and avoids redundant initialization computations. In doing so, EdgeOS guarantee that (1) an FWP is sealed so it cannot be modified after checkpointing; (2) an FWP is restored to the cached post-initialization state. Thus its memory is placed into a known and safe state, ensuring the integrity of future FWP instances. Checkpointing details are described in §4.3.

FWP Chains. To provide more complex functionality, FWPs can be arranged into chains, thus the entire chain can be efficiently managed as a whole. A FWP chain composes multiple checkpointed FWPs and are maintained in a *FWP-chain cache* that caches entire chains of FWPs, their interconnections, and their message pool. MMA copies data between adjacent FWPs within the same chain, avoiding shared memory.

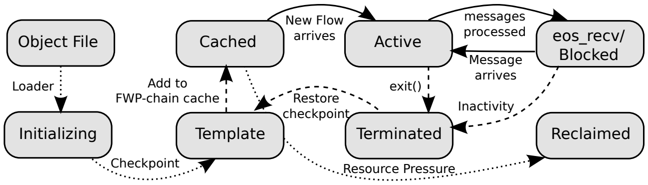


Figure 3: Lifecycle of a FWP-chain: Dotted lines indicate FWP manager operations conducted once to load and then checkpoint a FWP-chain, or to reclaim the FWP's resources when memory pressure exists. Dashed lines indicate operations to re-initialize terminated FWP-chains for future use. Solid lines are *data-path* operations performed by on the critical path.

3.4 EdgeOS Control Plane

The EdgeOS Control Plane is composed of: (1) the EdgeOS Controller that maps incoming flows to FWP chains, (2) the FWP Manager that controls the lifecycle of FWPs and optimizes their startup, and (3) the Scheduler that determines which FWP to run on each core and activates them in response to incoming messages.

Flow matching with the EdgeOS Controller. When new requests arrive from connected client devices, they need to be routed to the appropriate FWP chain. The EdgeOS Controller allows tenants to define FWP chains and the packet filtering rules that specify what traffic should be routed to them. These rules are pushed to the Net-In data plane component. Net-In applies rules similar to SDN match-action rules: packets are split into flows based on the header n-tuple (*e.g.* src/dest IP and port) and a rule is found that matches the flow. The rules indicate the FWP chain that will process that flow.[2] Since our focus is on fine-grained isolation and high scale, a rule can indicate whether all flows that match the rule should be handled by a single chain, or if each client flow should be given a dynamically started instance of the chain.

FWP Manager. Figure 3 illustrates the lifecycle controlled by the FWP Manager. Similar to a Linux process, an FWP starts as an object file, which must be loaded into memory. Once execution begins, FWPs perform some initialization routines, and are checkpointed to a Template. Then multiple identical copies are forked off the Template and put into FWP cache. As new clients arrive, they are paired with corresponding FWP-chains from the cache. The selected FWPs will be Activated, allowing them to process messages or transition to the Blocked state, before eventually Terminating when no longer needed. When a FWP chain terminates, the Manager reuses the chain by Restoring it to the post-initialization state and puts it back into the FWP-chain cache. If there is memory pressure, cached FWP templates and chains are Reclaimed.

[2] Our implementation currently assumes flow rules are statically preconfigured, but this could be extended to support on-demand flow lookups similar to SDN controllers, with a northbound interface to application logic that would assign a rule dynamically to each flow.

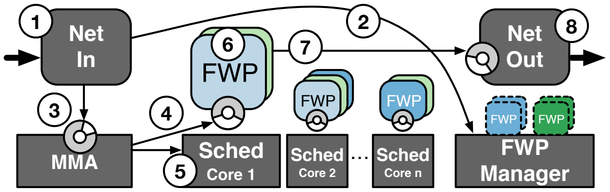


Figure 4: EdgeOS Timeline

Scheduling and inter-FWP coordination. Once a set of FWPs are activated, they are distributed across cores, and partitioned scheduling (*i.e.* without task migrations) multiplexes the core's processing time.

Traditional systems often use shared data-structures and Inter-Processor Interrupts (IPIs) for scheduling notification. For example, Linux activates threads by accessing that thread's data-structure directly to see if it is already awake, and if not, an IPI is sent. The resulting cache-coherency traffic and IPI overheads, can be significant, especially if used for message notifications arriving over a network at line rate. Motivated by these overheads, NFV platforms based on DPDK such as OpenNetVM [72] use active polling for communication between threads on different cores, thus avoiding blocking. However, as the number of processes ("network functions" in OpenNetVM) grows beyond the number of cores, spin-based event notification is inefficient.

All inter-scheduler coordination in EdgeOS is via message passing, avoiding shared memory synchronization. When a FWP-chain is activated, or when a message is sent to an FWP, the MMA notifies the scheduler of the activation. On the other hand, a FWP will be blocked after processing all of its messages. FWPs avoid spinning to ensure efficient multi-tenant computation. We currently use a simple and efficient preemptive, fixed-priority, round-robin scheduling policy. This aims to provide *temporal isolation* between untrusting FWP chains which prevents them from monopolizing the CPU, and from interfering with the progress of other tenants' FWPs.

Timeline Summary. Figure 4 shows the complete timeline for processing a packet. (1) A packet reception at the Net-In gateway causes a flow lookup to decide which FWP chain should process the packet. (2) If there is a miss, the FWP Manager spawns a FWP chain from its cache. (3) MMA copies the packet and (4) adds it to the first FWP's ring of the destination FWP chain. (5) MMA messages the scheduler on the FWP chain's core to activate it. (6) The FWP chain processes the packet and (7) the last FWP in the chain asks the output gateway to DMA the packet out the NIC.

3.5 Isolation Analysis

We summarize how EdgeOS achieves the isolation requirements listed in §1 based on our design principles. EdgeOS executes on top of a micro-kernel, with a smaller TCB than monolithic systems (on the order of 10K lines of code). FWPs access system resources through a capability-based access-control system provided by the kernel [67]. Capabilities protect and control access to kernel resources including synchronous and asynchronous IPC end-points, threads, time [20], and memory. The capability model is similar to that in seL4 [17], and relies on user-level retyping of untyped memory into both kernel resources and virtual memory. The most notable difference between EdgeOS's capability model and seL4's is that EdgeOS doesn't allow IPC-based delegation, instead relying on a user-level component with capability-based access to a FWP's capability-table to copy capabilities (thus access to kernel resources) into that FWP. EdgeOS leverages the kernel's capability system to tightly constrain FWP's access to system resources. Each FWP has access to only its own memory and to IPC endpoints to the scheduler, and to the FWP manager to block awaiting further execution, and expand their heap, respectively.

Trust model: §2.1 discusses the threats from untrusted tenant code, and from clients that can compromise that tenant code. As such, we assume that any resources available to an FWP will be used, where possible, to escalate privilege, and that all FWP system interfaces will be comparably stressed.

EdgeOS is implemented as a set of trusted, user-level components that have access to various FWP resources. The MMA has shared memory access to each FWP's message pool, and is trusted to properly move messages between adjacent FWPs, to and from the network gateways, and to notify schedulers of FWP activation. Similarly, the network gateways (using DPDK) are trusted to properly interface with the NIC and the MMA and to properly implement a tenant's flow matching rules. Per-core schedulers [54] have the ability to dispatch FWP threads, and are trusted to properly preemptively schedule FWPs and coordinate with the MMA to properly activate them. The FWP manager is relied on to quickly and correctly create new FWP instances, perform capability delegations into the FWP, maintain the FWP cache, and dynamically expand FWP heaps. The FWP manager delegates resources to FWP chains such that all readable/writable resources are partitioned per-FWP. Thus, each chain is mutually isolated, and when a chain is assigned to a client, clients are mutually isolated. Finally, the kernel is depended on to maintain the capability-based access model that constrains the set of resources available to each FWP.

FWP isolation: FWP memory is initially allocated by the FWP manager, and is of three types: (1) shared executable and read-only data derived from a tenant's FWP's image, (2) read-write memory in the global data segment and heap that is *not* shared among FWPs, and (3) message pools that are shared only with the MMA. Finally, each FWP has access to IPC end-points to request heap expansion, and to await the next message's arrival. Messages are sent downstream, and received from upstream FWPs using message pool, thus interfacing with the MMA using only simple wait-free ring buffers. The MMA maintains associations between an upstream FWP's message pool and the downstream FWP, thus only allowing data-flow within a chain. EdgeOS's design ensures that a malicious FWP – or a compromised FWP – will not be able to intercept data outside of the FWP's chain, nor impact the integrity of correct services.

Inter-FWP *temporal isolation* is enforced by preemptive scheduling. Each FWP executes in a separate thread controlled by per-core, round-robin scheduling logic. In contrast, many of the most efficient language-based techniques (e.g., NetBricks [52]) cannot prevent a buggy or malicious tenant's infinite loop from preventing progress for all tenants.

Inter-client isolation: Each client is served by a separate FWP (chain), the FWP (chain) is re-initialized before serving each new client, and when created, each FWP (chain) is delegated disjoint read/write resources by the FWP manager. Thus, a malicious client cannot impact the execution of future clients. In contrast, VM techniques often create a VM per-tenant [2], which executes multiple clients, thus potentially exposing clients to past compromises. Even webservers are typically architected to service multiple clients within a single protection domain.

When inter-client sharing is required by a tenant (*e.g.*, to implement a shared cache), FWP chains can either access network-accessible storage, or use Net-In gateway rules that send new clients to an existing FWP chain storing common state (§3.4). The former is similar to the stateless design of many serverless and microservice applications; in fact, FWPs provide more flexibility since per-user state can be maintained across multiple requests if desired. In the latter case, isolation is traded for sharing, which is evaluated in §5.6. More complex routing rules that cluster specific sets of clients into different, potentially existing, FWP chains are beyond the scope of this paper.

Per-client isolation in EdgeOS is, in many ways, most similar to systems to provide Distributed Information Flow Control (DIFC) [40]. Such systems track information as it flows (via IPC and other interactions) between processes, and define policies to determine when that flow is allowed. When a single process is needed in two conflicting information flows, a common strategy [15, 41, 62, 71] is to create a *new instance* per flow. This is similar in mechanism and motivation to the per-client FWP chain instantiation in EdgeOS. Importantly, EdgeOS focuses on providing instantiation of full FWP chains, low overhead (an order of magnitude less than Linux `fork`), and line-rate performance. Despite strong FWP isolation, EdgeOS achieves *per-packet* overheads on the order of dedicated middlebox infrastructures that do not provide comparable isolation.

4 Implementation

We implemented EdgeOS in Composite (`composite.seas.gwu.edu`), an open source μ-kernel that externalizes traditionally core kernel features into user-level *components* that define the resource management and isolation policies [67]. In Composite, components interact through highly-optimized Inter-Process Communication (IPC) to leverage system logic and resources. Composite is based on a capability-based protection model [17, 61] that controls component access to kernel resources. The kernel includes no scheduling policies, instead implementing schedulers at user-level [54]. The Composite kernel scales well to multiple cores as it has no locks and is designed entirely around store-free common-paths, wait-free data-structures, and quiescence [67]. MMA can be implemented in other OSes such as Linux. In EdgeOS we pair it with the FWP abstraction to provide fine-grained isolation and adaptability to churn.

EdgeOS prototype consists of the MMA, FWP management, DPDK-based network access and schedulers. In total, EdgeOS adds fewer than 6000 lines of code. We plan to release our code and experiment templates for repeatable research.

4.1 Message Pool Management

Memory management integration into ring-buffers. Each FWP's message pool is associated with two ring buffers that track *both* how to transmit and receive messages, *and* the allocation and deallocation of messages. EdgeOS observes that general purpose memory allocation facilities (`malloc/free`) can have significant overhead. Thus, we integrate memory with message management by tracking free memory in rings.

A reception ring buffer contains a set of references to message slots into which incoming data can be copied, and the transmission ring buffer contains references to messages to move downstream in the FWP chain. The MMA orchestrates data movement between different FWP's packet memory regions, thus acting as a software DMA accelerator.

Message pools are managed by FWPs as a span of MTU-sized message slots, and unlike traditional NIC DMA ring buffers, the ring buffers include an entry for *each* message slot. Ring entries that have been transmitted by an FWP, and have been copied by the MMA are marked as free, and are used for packet allocations. FWPs must maintain a sufficient number of messages in reception rings to buffer messages that queue up due to the system's scheduling latencies. Thus, after FWPs finish processing pending messages, they move batches of freed messages from the transmit ring into the reception ring. This avoids `malloc` on the fast path, as message *liveness* is managed indirectly through the ring buffers.

Message pools and isolation. The ring buffer design decouples the *message pool* from the *meta-data* to coordinate the data movement and liveness between FWPs and the MMA. This avoids lock-based protection of the rings, instead relying on wait-free mechanisms. This is necessary to avoid the high costs of synchronization, and ensure progress of the MMA in spite of possibly malicious FWPs.

4.2 Memory Movement Accelerator

Our initial experiments showed that naively copying packets in a DPDK-based NFV pipeline decreased throughput by more than 50%. However, a MMA core has a throughput of around 54 Gb/s on our hardware, which is sufficient for line-rate. For networks that require a higher throughput, more cores can be specialized as MMAs. In the limit, MMA's throughphut is bounded by the chip's memory bandwidth, which for our processor is 472 Gb/s. By using the parallelism of the underlying processor and specializing cores to run the MMA, we achieve both isolation and high throughput by taking message movement out of the critical path.

The MMA has read-write access to all message pools. It maintains a mapping between both pairs of transmit and receive ring buffers for subsequent FWPs in a chain, and continuously iterates through all such pairs, transferring messages when it finds a transmitted message. The MMA provides two essential services: *data-movement by copying transmitted messages*, and *event notification of the receiving FWPs*. The MMA's FWP event notification is efficient as it simply sends a message to the scheduler controlling the target FWP's, and relies on the scheduler to asynchronously process events.

MMA optimizations. As the MMA is on the data-path of all FWP interactions, including message reception, it must be able to move messages at faster than line rate. The data-structures linking transmit and reception rings are laid out in an array to leverage the processor's prefetcher as the MMA iterates over them. The initial implementation of the operations on the ring buffers were straight-forward, but cache-coherency traffic, possibly a cache-line transferred for each ring entry, hurt throughput. To address this, we optimize the MMA:

- Double-cache-line (128B) *caches* are added to both the enqueue and dequeue operations. These caches are in local memory outside of the ring, thus their modifications avoid coherency traffic. When retrieving to the ring, a batch is copied into the cache, and when transmitting messages are queued in the cache, and batch copied into the ring. To ensure message delivery, the cache is flushed by an FWP before it blocks.
- These caches enable messages to be transferred in batches. We use explicit software prefetch instructions to load all referenced messages in the cache to avoid CPU cache misses on message processing.
- Messages are efficiently addressed and copied as the MMA has shared memory access to all message pools. To maintain protection, the MMA validates that FWP messages are within a valid message pool.

4.3 Optimized FWP checkpointing

EdgeOS caches the images of *chains* of FWP binaries so they are ready for prompt activation. These ready-to-execute images are *asynchronously* prepared, thus moving the overhead for FWP preparation off the fast-path. The cached FWP's state is identical to the *initialized* state of a ready-to-execute FWP.

We utilize a few optimizations to efficiently generate post-initialization FWP snapshots: (1) the post-initialization checkpoint of the FWP-chain is laid out contiguously in memory so that chain re-initialization is as close to `memcpy` / `memset` overheads (for which we use the `musl` libc, unoptimized versions), (2) we do not eagerly reclaim – and thus later re-allocate – heap memory from terminated FWPs, instead only zeroing it out to maintain confidentiality, and using it to satisfy future heap allocations, (3) we reuse the threads active in each FWP by only resetting their registers to the appropriate post-initialization state, which avoids the overhead of thread destruction and allocation, and (4) only if there is memory pressure do we reclaim first spare FWP heap memory, then cached FWPs. Re-initialized FWPs maintain zero state from their previous execution: the stack, heap, and writable data sections are reset to the initial state. These optimizations culminate in a system that can handle exceedingly high churn and scalability: FWP chain initialization is dominated by `memcpy`/`memset` overheads, and new client chain activation takes in the low 10s of μ-seconds.

5 Evaluation

All experiments are run on CloudLab Wisconsin c220g1 series nodes [57]. These are 2 socket, 8 core, Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz processors with 128GB ECC Memory. Note that these systems have fewer cores than current edge offerings, thus pressuring EdgeOS's design that dedicates cores to different functions. Systems are connected via Dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes).

5.1 Latency and Throughput

We first evaluate the latency and performance predictability of EdgeOS compared to other high performance networking platforms. Figure 5(a) shows the response time distribution (in microseconds) for an ICMP ping response Click [28] element implemented as either: a DPDK process, an OpenNetVM NF (ONVM), a standard linux process with kernel-based IO, a ClickOS NF in a Xen VM, or an FWP in EdgeOS. The results show that EdgeOS significantly outperforms all of these techniques (by up to 3.8X in average latency), except for DPDK. DPDK is slightly better because it can run only a single service at a time and thus does not need to copy packets from the initial receive DMA ring to a separate pool. In contrast, EdgeOS provides a platform to potentially run thousands of distinct services, and thus needs to offer stronger isolation via copying.

Figure 5(b) shows the maximum throughput of different approaches when forwarding traffic from `pktgen`, a high speed packet generator. EdgeOS again provides better performance than ClickOS, while offering stronger isolation than DPDK and ONVM, which rely on globally shared memory pools for zero-copy IO.

Next we compare the performance of EdgeOS communication with ONVM. We run a chain of NFs on the same core that each forward small (64B) or big (1024B) packets, thus both

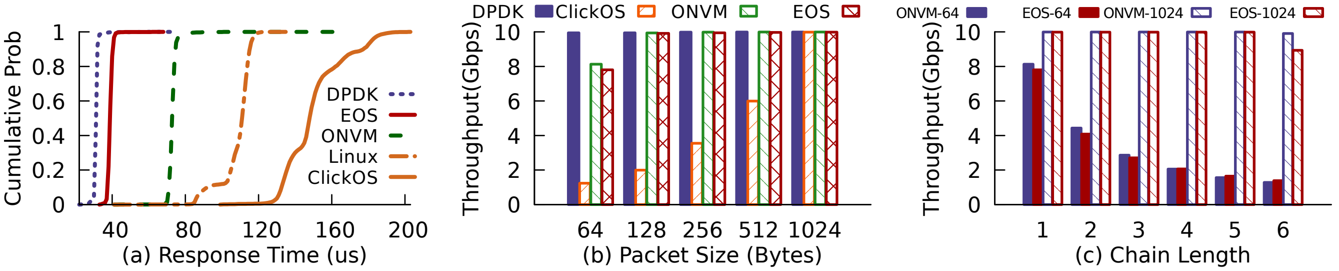


Figure 5: (a) EdgeOS provides substantially better latency, and reduced jitter compared to Linux processes and NFV platforms like OpenNetVM and ClickOS. (b) Throughput of each system with different packets sizes. (c) EdgeOS provides isolation and adds negligible overheads compared to OpenNetVM (no isolation) for different chain length for messages of size 64 and 1024 bytes.

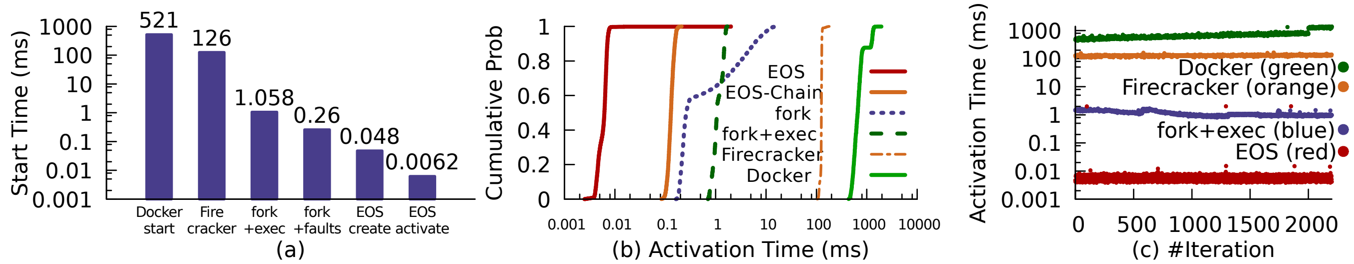


Figure 6: EdgeOS provides orders of magnitude better startup time than other approaches and does not suffer from scalability problems when starting larger numbers of FWPs.

systems have context switch overhead by passing a packet to the next NF. In addition, EdgeOS has copying overhead from the MMA to enforce isolation. The results in Figure 5(c), show that as the chain length increases, the throughput of 64B packet drops for both EdgeOS and ONVM. The main overhead of EdgeOS is data copying, while the overhead of Linux context switches and scheduling dominates ONVM. When the chain length is smaller than 3, the overhead of copying is less than 8%, and EdgeOS outperforms ONVM when the chain is longer as the Linux system overheads increase. The throughput with 1024B packets maintains line rate for both systems when the chain length is smaller than 6, at which point EdgeOS sees a throughput decrease. Even at length 6, the MMA is able to maintain an aggregate of 54 Gb/s.

5.2 Startup Time

FWP Initialization and Activation. In Linux, initializing a process involves calling `fork` (and possibly `execve`). For Docker containers, a `docker run` command is similar, but includes additional system calls to configure namespaces and maintain container metadata. For Firecracker, we use the recommended "hello" image and use 1 vCPU and 128 MiB RAM. In order to optimize the fast path of readying a cached FWP, EdgeOS separates out creation from activation. For EdgeOS, creation involves transitioning from the Object File to Cached state in Figure 3, including setting up page tables, capability tables, and thread creation. We record the start time for 10,000 iterations of starting a container, VM, process, or FWP and report the median in Figure 6 (a). Note the log scale. We use median time values as Container creation cost increases slowly over time so the mean is skewed by these outliers. We compare against two variants of Linux processes: "fork + exec" loads a different binary whereas "fork + faults" mimics loading the service's working set by issuing writes to eight different pages to trigger page faults (the size of the minimal FWP). These approaches are 5-20X slower than the comparable "EOS create" approach (dashed lines in Figure 3).

Once an FWP has been created, EdgeOS keeps copies of it in a cache which can be quickly activated on demand (solid lines in Figure 3). Cached activation improves EdgeOS performance by another order of magnitude, allowing new processing entities to be instantiated in 6.2 microseconds. Figure 6(b) presents a CDF of these approaches, including the activation cost for a full chain of 10 isolated FWPs, which remains an order of magnitude faster than fork+exec.

FWP Scalability and Middlebox Computation. Containers and VMs suffer from poor scalability: as the number of instances rise, the start time increases [32]. In Figure 6(c) we show the time to start a new container, create a Firecracker VM, exec a process, and activate an FWP, when up to 2200 are started incrementally. The Container case gradually drifts upward before hitting a step after 2000 containers (note log-scale) – the last container takes 1.368 seconds versus 0.467 seconds for the first. FWPs provides nearly constant start time regardless of scale. EdgeOS has a few outlier points (11 out of 15K measurements are at 2ms), which we believe to be Non-Maskable Interrupts, or a bug in our scheduling logic.

5.3 Isolation

Just in Time Service Instantiation. To evaluate the impact of client churn in edge environments, we measure client response time for a ping that creates a new FWP. Clients send requests at a configurable interval, and we assume that each new client requires a new, isolated FWP. The new FWP re-

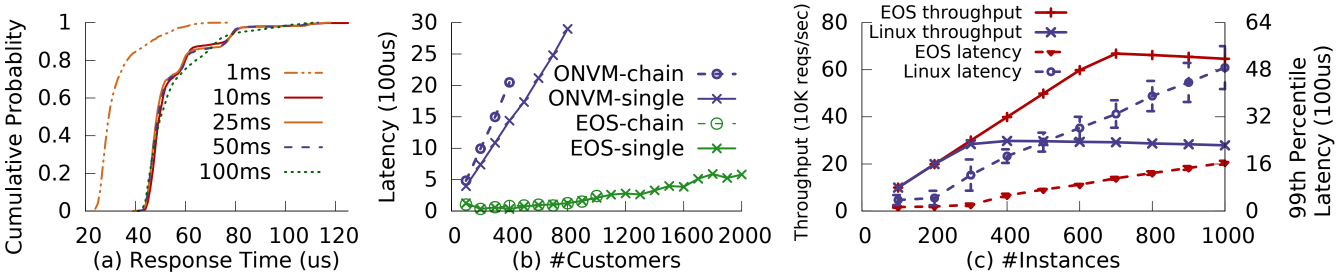


Figure 7: (a) EdgeOS just in time service instantiation for each for mobile client connection with varying client inter-arrival rates; (b) Routing and processing latency for middlebox routing `netperf` traffic for an increasing number of clients; (c) TLS termination performance for up to 1000 end points (solid lines: throughput, dashed lines: tail latency).

ceives the incoming packet, produces a reply, and then terminates, representing a worst case churn scenario. Figure 7(a) shows a response time CDF for EdgeOS under different client arrival patterns. The results show that even when a new client arrives every millisecond, 90% of requests are serviced within 50 microseconds This experiment mimics that in LightVM [32], and although we have not been able to successfully run the LightVM software on our testbed, we note that their paper produced a 90th percentile response time of 20 milliseconds (more than 400X worse) with 10ms client arrivals. The EdgeOS performance advantage comes from our extremely lightweight FWP abstraction and our template cache that allows nearly instant instantiation.

Multi-Tenancy and Customer Isolation. An important job of edge-cloud systems is to act as a middlebox to monitor traffic close to the source. Figure 7(b) depicts the processing latency of a middlebox deployed between `netperf` client and server machines for an increasing number of concurrent clients. We use three nodes, two running `netperf` clients and servers, and the third running EdgeOS or ONVM in the middle. The systems run either a single firewall to filter flows or a 2 FWP chain of firewall plus monitor, all implemented in Click, to further maintain statistics about flows. Each customer is serviced by its own "personal firewall" or chain, thus preventing malicious clients interfering with others. We measure the middlebox latency overhead (i.e., the added cost versus direct client/server connections from Figure 1) as we increase the number of clients, and thus number of FWPs (EdgeOS) and Network Functions (NFs in ONVM).

Though ONVM is a highly optimized middlebox infrastructure, it relies on containers and expensive coordination mechanisms between NFs and the management layer. Because of this, ONVM cannot scale past around 820 containers or 410 chains, and the added latency rises quickly with each new client. FWPs enable the system to scale past 2000 clients with an average increase in the latency of only around $0.3\mu s$ per additional client. Chaining in EdgeOS adds negligible latency overhead thanks to efficient FWP scheduling and activations, while ONVM sees an increasing gap since it relies on Linux's more heavyweight futexes.

5.4 TLS Termination

For our first edge cloud use case, we consider the deployment of edge-based TLS termination proxies, such as for a CDN serving `https` traffic or for IoT devices sending encrypted data streams. This requires an edge end-point for TCP connections, a TLS handshake to share public keys, and continued encryption/decryption of transferred contents. As the majority of web traffic is over `https` [59], TLS implementations are a high-priority target for compromises, and have a history of high-impact vulnerabilities, e.g., Heartbleed [12]. Therefore, instead of sharing one `https` end-point among many clients, we instantiate an isolated TLS FWP for each client. Toward this, we ported `axtls` (`axtls.sourceforge.net/`) (version 2.1.4), which includes a lightweight `https`-based web proxy optimized for embedded systems, and the `lwip` (`savannah.nongnu.org/projects/lwip/`) TCP/IP networking stack (version 2.1.2) to EdgeOS.

In our experiment we use a single cloudlab node as the edge server, and use five additional nodes to drive the client workload. We style this experiment after the setup in [32], and request zero-length files hosted at the proxy (headers are still encrypted). Each client uses ab (version 2.4.39) and keep-alive sessions to make a series of requests over a TLS-encrypted session. We modified ab by adding a `nanosleep` to rate limit each client to 1000 requests per second. We disable Nagle's Algorithm for these experiments since it leads to very low throughput and low network utilization due to an adversarial interaction with delayed ACK support. For fairness, `axtls` on Linux stores files in a ramdisk.

Figure 7(c) depicts the results of running an increasing number of clients making `https` requests. Both Linux and EdgeOS saturate the CPU at around 350 and 700 clients and reach 297K and 668K requests per second, respectively. Similarly, EdgeOS achieves around three times lower 99th percentile latency than `axtls`, and has lower variability across clients compared to Linux as shown by the error bars.

In addition to fast FWP instantiation, and efficient communication, the following FWP optimizations are significant: (1) the FWP abstraction focuses on communication with a *single* client, it avoids event multiplexing through `select` and the associated overhead, and (2) similarly, the share-nothing nature of the FWP abstraction enables synchronization-free

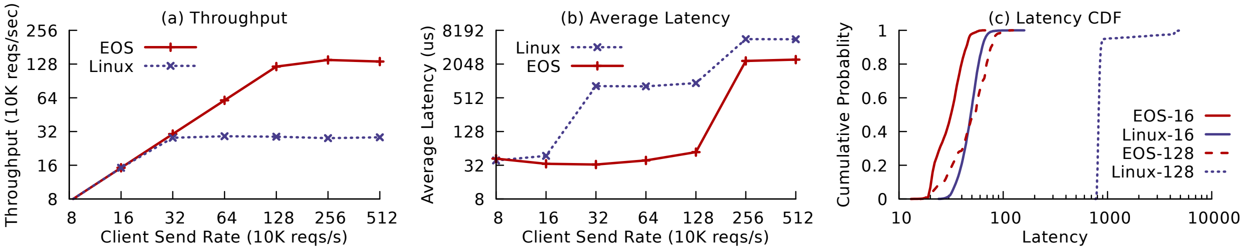


Figure 8: Single memcached instance on one core.

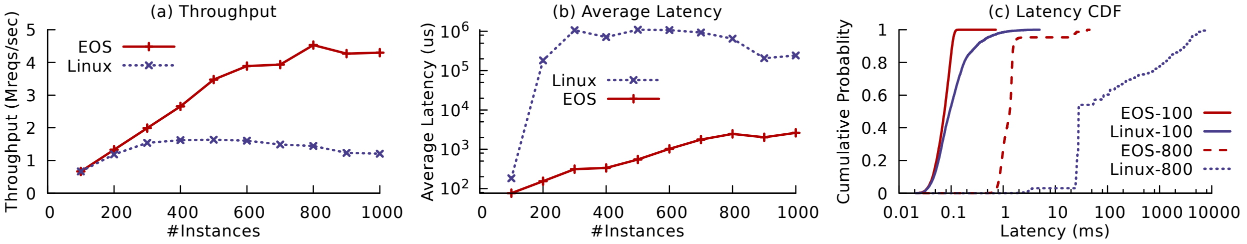


Figure 9: Multiple memcached instances (one per tenant) on 16 cores.

networking using the simple `lwip` stack (in contrast to some unikernels that cannot push `lwip` to Linux-level throughput [32]). Though EdgeOS provides significantly stronger isolation (a TLS instance per client), the FWP model still enables efficient, predictable scalability.

5.5 Edge Inference

Edge based neural network inference enables resource constrained embedded systems to offload computationally-heavy work such as live image recognition. For our second use case we port the CMSIS NN Library (version 1.0.0) neural network inference library to EdgeOS. We use the example CIFAR-10 configuration which takes as input a 32x32 pixel color image which classifies to number 0-9. We focus on providing each tenant with isolated inference services. Thus, we compare an EdgeOS CMSIS NN FWP-per-client against a simple Linux server that `fork`s a process for each client.

	Linux Clients		EdgeOS Clients	
	100	500	100	500
Mean latency (ms)	13.8	69.2	14.6	70.1
99th percentile (ms)	25.4	135	14.8	71.3

We utilize a PowerEdge R740 server with two 24-core Intel Xeon 8160 sockets that represents the Amazon and HPE edge offerings. We use either 100 or 500 clients (separate columns), each requesting 500 inferences. This application is particularly CPU-heavy (each inference taking around 6ms), thus penalizing EdgeOS' design that specializes cores, i.e., Linux can use all cores on the system for inference, whereas EdgeOS sets aside 4 cores for MMA and control services.Despite this, EdgeOS has only slightly lower throughput than Linux, losing 2.5% to 4.6%. However, the simpler FWP runtime in EdgeOS minimizes scheduling interference, reducing latency jitter. Together with EdgeOS's more efficient activation, this yields significant decreases in tail latency – 42% and 47% at 100 and 500 clients, respectively.

5.6 Memcached

Finally, we evaluate how EdgeOS can provide a platform for low latency endpoint applications that don't require rapid instantiation. We implement memcached as an FWP that uses UDP for requests. We mimic a scenario where one or more edge tenants store data, each with many clients making requests. Isolating these tenants from each other is necessary as they should not be able to access (maliciously or not), other tenant's cached data. The EdgeOS controller is used to map incoming requests either to a single memcached FWP (e.g., representing a typical edge cloud data cache) or one FWP per tenant (e.g., representing data stores for different sets of edge-connected IoT devices). We compare EdgeOS against Linux, with a single, or multiple memcached instances. Our workload uses 135 byte value sizes and a 95% get, 5% set request mix generated by the mcblaster client as in [46]. We use multiple 16-core client machines, each running mcblaster processes to ensure the client will not be a bottleneck.

Figure 8 shows a single memcached instance while Figure 9 shows a variant number of instances, representing different edge-cloud tenants. We report the aggregate throughput across all requests, the average latency, and a CDF of the latency (with 16K or 128K clients) to understand the tail. The single instance focuses on the data-path efficiency of the system, while the multi-instance evaluates each system's scalability to an increasing number of tenants on the limited edge hardware.

The efficiency of both systems is seen in their aggregate throughput. EdgeOS processes over 5x the throughput for a single instance, and can scale more gracefully up to 800 instances whereas Linux handles up to 400. EdgeOS' response time for a single memcached instance is substantially lower than Linux: it handles 8X the client request rate before seeing an increase in latency. Since Linux is not able to keep up, it drops a large number of requests, e.g., 5.2% at a 320K req/sec client rate. In contrast, EdgeOS does not see any requests drops at a 1.2M req/sec client rate. For multiple instances, Linux has a response time of nearly 1 second, whereas EdgeOS has an average latency below 1 millisecond for up to 600 memcached instances. From the latency CDF, we observe that even with only 100 memcached instances, Linux has much higher tail latency than EdgeOS, and that with 800 instances Linux has more than three orders of magnitude worse tail latency. These latency metrics ignore dropped requests – with 800 instances, EdgeOS drops 13% of requests, whereas Linux drops 66%.

6 Related Work

Multi-tenant isolation. Significant research addresses isolation in a multi-tenancy environment. Bolted [38] presented an architecture for a bare metal cloud supporting security sensitive tenants. PSI [70] enables fine-grained and dynamic security postures for different network devices by assigning each device an NF. Denali [68] separates the protection provided by a Virtual Machine Manager (VMM) from the abstractions within a VM, and enables lightweight VM contexts. Multi-tenancy virtual switch designs are proposed in [60, 64]. In contrast, EdgeOS is motivated by the potentially enormous churn and large-scale isolation requirements of the edge cloud, providing service to transient mobile and IoT devices. For isolated edge computation instantiation, FWP compares favorably to forking of minimal Linux processes (two orders of magnitude faster start-time) which is the lower-bound for many such techniques. Re-initializing FWPs to safe states is partially motivated by ChaosMonkey [3, 5].

Lightweight isolation. Wedges [6], LWC [29], and SpaceJMP [16] expand the UNIX interface to include lightweight facilities for controlling and changing protection domains. Similarly, Dune [4] uses hardware virtualization support to provide user-level control over page-tables, and both dIPC [65] and Skybridge [36] use hardware support to bypass the kernel during inter-protection domain communication. Several projects have increased the efficiency of containers. Cntr [63] includes only the application-specific context in a container, while SOCK [48] specializes the container to use efficient kernel operations, and uses a Zygote mechanism paired with a cache to accelerate container creation for stateless computations. EdgeOS targets at abstractions to support immense churn rates, efficient communication with strong isolation via the MMA and a narrow system attack surface. To efficiently use the limited resources in the edge cloud, EdgeOS leverages this support to scale to more than two thousand FWPs in less than 1GB of RAM while maintaining line-rate communication.

Other isolation mechanisms. DMA shadowing [33] utilizes extra memory copies for DMA buffer to provide full IOMMU protection. DAMN [34] introduces DMA-aware packet memory allocator to achieve efficient IOMMU protection. MMA also uses data-copying to avoid shared memory communication between untrusted FWPs. LXDs [42] runs isolated kernel subsystems on dedicated cores. EdgeOS achieves similar isolation with the micro-kernel approach. EdgeOS implements its system-level services in user-level, and further spreads them to different cores.

New hardware features are used to enhance memory isolation, such as Intel MPK [25, 53] and SGX [50, 56]. They are complementary to EdgeOS. Language techniques such as NetBricks [52] implement network processing functions in a memory-safe language. These techniques rely on software isolation within a *single thread*. Without multiplexing the CPU among untrusted FWP chains via preemptive scheduling, *temporal isolation* is challenging. EdgeOS effectively uses the MMA to maintain memory safety, but also provides temporal isolation by executing all FWPs in separate threads that are preemptively scheduled. We also support the direct execution of legacy code modulo the confines of FWP APIs.

7 Conclusions

The increasing prevalence of mobile computations and the Internet of Things requires both scalable isolation facilities for multi-tenancy in the edge, and the agility to handle high churn. This paper has described an optimized copy-based MMA architecture that provides strong mutual isolation without performance penalties. We introduced FWP for scalable isolation that is paired with a cache of post-initialization checkpointed FWP-chains to provide microsecond scale activation times for high churn.

Our evaluation shows EdgeOS substantially improves performance for a wide range of applications from network middleboxes to endpoint services. We show that EdgeOS provides more than a 3.8X reduction in ping latency and more than 2X throughput increase compared to ClickOS – a system that also provides isolated computation – for middlebox computations. More importantly, EdgeOS can create FWPs for client computation in 25-50 microseconds, even when they are created every millisecond, and can scale to over 2000 FWPs while maintaining low latency, even with a very limited amount of memory. For edge applications like memcached, EdgeOS has more than three orders of magnitude decreases in latency when running over 300 server instances simultaneously, and even CPU-intensive TLS termination shows a factor of three tail latency decrease, all while maintaining strong isolation. We believe that EdgeOS paves the way for closely integrating the edge cloud into – and augmenting the capabilities of – the increasing prevalence of mobile and embedded devices.

Acknowledgments. We would like to thank the anonymous reviewers, especially our shepherd Trent Jaeger, for their tremendous feedback that has significantly improved the quality of this paper. We are also thankful to Phani Kishore Gadepalli, Zheng Yang and Runyu Pan for their help on the Composite system. This work was supported by the National Science Foundation under Grants CNS 1815690, CNS 1814234, and CPS 1837382.

References

[1] 5g network slicing in 5gtango, `https://www.5gtango.eu/blog/36-5g-network-slicing-in-5gtango.html`, 2019.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, 2020.

[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, USA, October 8-10*, 2012.

[5] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. *Netflix Tech Blog*, 30, 2012.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[7] William Earl Boebert and Richard Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[8] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[9] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 26(1):29–35, 1983.

[10] Docker: https://www.docker.com/, 2018.

[11] Intel Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM. event-place: Vancouver, BC, Canada.

[13] Telecommunications industry association. edge data centers. `https://www.tiaonline.org/wp-content/uploads/2018/10/TIA_Position_Paper_Edge_Data_Centers-18Oct18.pdf`, 2018.

[14] Micro-data centers out in the wild: How dense is the edge?, `https://www.datacenterknowledge.com/archives/2017/05/02/edge-densities`, 2017.

[15] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[16] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[17] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[18] Dawson R. Engler, Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, USA, December 1995. ACM.

[19] Firecracker: https://firecracker-microvm.github.io/, 2019.

[20] Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. Temporal capabilities: Access control for time. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017.

[21] The 5g guidea reference for operators the 5g guide: A reference for operators, `https://www.gsma.com/wp-content/uploads/2019/04/The-5G-Guide_GSMA_2019_04_29_compressed.pdf`, 2019.

[22] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, 2019.

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[24] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[26] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 341–354, New York, NY, USA, 2007. ACM.

[27] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, April 2018. USENIX Association.

[28] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[29] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[30] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[31] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11), December 2013.

[32] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[33] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 249–262, New York, NY, USA, 2016. ACM.

[34] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. Damn: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 301–315, New York, NY, USA, 2018. ACM.

[35] Bishop Matt et al. *Introduction to computer security*, volume 50. Pearson Education India, 2006.

[36] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[37] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, Mountain View CA (USA), 2003.

[38] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting security sensitive tenants in a bare-metal cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[39] Shahid Mumtaz, António Morgado, Kazi Huq, and Jonathan Rodriguez. A survey of 5g technologies: Regulatory, standardization and industrial perspectives. *Digital Communications and Networks*, 2017.

[40] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

[41] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical difc enforcement on android. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[42] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. Lxds: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[43] NGMN Alliance, 5G End-to-End Architecture Framework, 2017.

[44] NGMN Alliance, 5G White Paper, 2017.

[45] NGMN Alliance, Description of Network Slicing Concept, 2017.

[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[47] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 1–14, New York, NY, USA, 2017. ACM.

[48] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[49] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira. Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges. *IEEE Communications Magazine*, 55(5):80–87, 2017.

[50] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[51] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[52] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[53] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019.

[54] Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the Composite component-based system. In *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS'08), Barcelona, Spain, November 30 - December 3*, 2008.

[55] Larry Peterson. Cord: Central office re-architected as a datacenter. *Open Networking Lab white paper*, 2015.

[56] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[57] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[58] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker. Network slicing to enable scalability and flexibility in 5g mobile networks. *IEEE Communications Magazine*, 2017.

[59] Sandvine. The Global Internet Phenomena Report, October 2018.

[60] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[61] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99), Kiawah Island Resort, South Carolina, USA, December 12-15*, 1999.

[62] Yuqiong Sun, Giuseppe Petracca, Xinyang Ge, and Trent Jaeger. Pileus: Protecting user resources from vulnerable cloud services. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (CCS)*, 2016.

[63] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[64] Kashyap Thimmaraju, Saad Hermak, Gabor Retvari, and Stefan Schmid. MTS: Bringing multi-tenancy to virtual networking. 2019.

[65] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems (Eurosys)*, 2017.

[66] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53. ACM, December 1995.

[67] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. Speck: A kernel for scalable predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15), Seattle, WA, USA, April 13-16*, 2015.

[68] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications, 2002.

[69] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[70] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. PSI: precise security instrumentation for enterprise networks. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[71] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008.

[72] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM, August 2016.

Firefly: Untethered Multi-user VR for Commodity Mobile Devices

Xing Liu   Christina Vlachou   Feng Qian   Chendong Wang   Kyu-Han Kim**
*University of Minnesota, Twin Cities   *Hewlett Packard Labs*

Abstract

Firefly is an untethered multi-user virtual reality (VR) system for commodity mobile devices. It supports more than 10 users to simultaneously enjoy high-quality VR content using a single commodity server, a single WiFi access point, and commercial off-the-shelf (COTS) mobile devices. Firefly employs a series of techniques including offline content preparation, viewport-adaptive streaming with motion prediction, adaptive content quality control among users, to name a few, to ensure good image quality, low motion-to-photon delay, a high frame rate at 60 FPS, scalability with respect to the number of users, and fairness among users. We have implemented Firefly in 17,400 lines of code. We use our prototype to demonstrate, for the first time, the feasibility of supporting 15 mobile VR users at 60 FPS using COTS smartphones and a single AP/server.

1 Introduction

Virtual Reality (VR) has registered numerous applications. In this paper, we focus on *multi-user VR* where multiple users jointly participate in exploring a VR scene. This enables many applications that single-user VR cannot support such as team training, social VR, group therapy, collaborative product design, and multi-user gaming.

We envision the following use case with more than 10 collocated users in a VR room. To start multi-user VR, each user simply launches the app on her smartphone and plugs the phone into a VR headset (*e.g.,* a $50 Samsung Gear VR [18] or even a $10 Google Cardboard [9] with a $6 VR controller [5]). These mobile devices fetch the VR content from an off-the-shelf server based on the users' real-time motion. The devices and the server communicate wirelessly over a single WiFi access point (AP). The users can enjoy the high-quality VR content as if it is rendered by a desktop PC with a powerful GPU. Meanwhile, each user can see and possibly interact with other users in the virtual world.

This paper aims at realizing the above ambitious use case. We design and implement Firefly, a novel multi-user VR system for mobile devices. The goals of Firefly are the following. First, Firefly works with affordable, commercial off-the-shelf (COTS) mobile devices, server, and AP. This helps reduce the deployment cost and facilitate the "bring-your-own-device" (BYOD) policies that many enterprises adopt today [6]. Second, Firefly employs untethered, wireless VR to overcome the inconvenience and trip hazards incurred by wired cables [19]. This is important for multi-user VR where multiple users' cables may easily get intertwined. Third, Firefly offers high content quality, low "motion-to-photon" (M2P) latency, and a high frame rate. An M2P higher than 16ms can cause nausea to VR users [11]. We target Quad HD (1440p) resolution, 60 frames per second (FPS) that can provide a good experience even for fast-paced VR gaming – the most demanding VR task [10]. Fourth, Firefly aims at supporting ∼15 users who can form a sizeable group of, for example, co-workers, students, or patients. To our knowledge, no existing system can achieve this using a single commodity server and WiFi AP. Recent work on multi-user VR only demonstrated 4 concurrent emulated users [47]. Fifth, Firefly allows complex VR scenes with both background and dynamic foreground objects, such as other users' avatars that users can interact with.

The above goals pose multiple challenges. The CPU/GPU power of a smartphone is at least one order of magnitude lower than its desktop counterpart [57], not to mention the energy/heat constraints; the heterogeneity of their computational capabilities should also to be taken into consideration; the bandwidth offered by a single AP is limited for multiple users; another key challenge is multi-user scalability, which calls for strategic decisions of splitting the client-server workload, as well as scalable approaches for rendering and distributing the content. To address the above challenges, Firefly makes a series of judicious design decisions as follows.

• Firefly performs one-time, offline content preparation by enumerating, pre-rendering, encoding, and storing the views at all positions reachable in a virtual scene [27]. At runtime, given a user's position and viewing direction, the server directly retrieves the stored high-quality content and delivers it to the user. This completely eliminates the online rendering overhead. Prior work [27] applies offline rendering to a single mobile device for local VR scenes, while Firefly further extends this concept to networked multi-user VR where offline rendering is found to be an indispensable mechanism ensuring scalability (§3.1).

• To reduce the network bandwidth consumption, Firefly takes a viewport-adaptive approach: each user only requests for the content that the user is about to perceive based on motion prediction. We conduct a thorough analysis of 25 human users' motion traces collected from an IRB-approved user trial. The results shed light on developing a lightweight yet effective motion prediction approach for Firefly. In the literature, several studies [24, 33, 39] have examined 360° video viewers' viewing patterns that only involve rotational movement (yaw and pitch). Our study instead investigates generic VR users' motion that consists of both the rotational and translational viewport movement as well as their interplay (§3.2).

- Firefly supports *Adaptive Quality Control (AQC)*, which determines the content quality of each user based on the total network bandwidth, the bandwidth available to each user, and the amount of to-be-delivered content. AQC essentially extends traditional video bitrate adaptation [40, 41, 51, 66]: from handling a single client to multiple clients, from dealing with regular videos to immersive VR content, and from being invoked at the second level to the millisecond level to adapt to users' motion. These differences require AQC to be effective, lightweight, fair, and scalable as reflected in our design (§3.4).
- Firefly handles dynamic foreground objects in a scalable and adaptive manner. Specifically, objects' 3D models are distributed to the clients offline. They are then rendered locally by the client. This eliminates the uncertainty caused by the network as well as the potential resource competition from other users compared to a server-side approach. To prevent too many objects appearing in the viewport from slowing down client-side rendering, Firefly supports adaptively reducing the objects' fidelity to maintain a high FPS (§3.6).

Additionally, Firefly has integrated several system-level optimizations, such as motion prediction error toleration (§3.3), client-side hierarchical cache (§3.5), and AP-assisted bandwidth estimation (§4). Our implementation on commodity Android/Linux platforms involves 17,400 lines of code. We conduct extensive evaluations using commercial VR scenes, real users' motion traces, and off-the-shelf smartphones/AP/server. We highlight the evaluation results as follows (§5).

- Firefly achieves very low motion-to-photon delay (≤15ms for 99% of the frames), low stall duration (around 1 second per minute), a frame rate at 60 FPS, and fairness among the users when supporting 15 concurrent users with a single server and a single 802.11ac AP (§5.2).
- Firefly is adaptive to users dynamically joining and leaving the system as well as network bandwidth changes (§5.4,§5.5).
- Firefly significantly outperforms existing systems. We extend Furion [44], a state-of-the-art single-user VR system over WiFi, to support multi-user VR. Due to its more efficient content fetching strategy, Firefly exhibits 18% higher median FPS, 6.9× lower stall duration, and much higher content quality, compared to multi-user Furion (§5.2). We also use our 15-user dataset to evaluate MUVR [47], a very recently proposed multi-user mobile VR framework. Through simulation, we find that for 27% of the time, the MUVR server still needs to perform online rendering for more than 5 devices. This makes MUVR not scalable to many users (§5.6).
- Firefly incurs acceptable CPU, GPU, and memory usage. When tested on 5 modern smartphones, after 25-minute VR sessions, the battery life percentage drops by 4% to 8%, and the devices' temperature reaches no higher than 50°C (§5.7).

Firefly is to our knowledge the first system that can scale untethered multi-user mobile VR. We make multi-fold contributions in this work: (1) the design of Firefly, (2) the study of real VR users' motion, and (3) our prototype implementation that demonstrates the support of 15 VR users at 60 FPS using COTS smartphones and a single AP/server. With emerging wireless technologies (*e.g.*, 802.11ax and 5G), we believe that Firefly has the potential to scale up to even more users.

2 Motivation and Overview

Firefly enables multiple users (10+) to simultaneously enjoy high-quality VR at 60 FPS using commodity smartphones, a single off-the-shelf server, and a single WiFi access point. We consider three high-level architectural design options.

A Serverless Design does not involve a server, so all the VR content is stored on users' mobile devices, which also perform full-fledged rendering. Most of today's commercial 3D games and VR mobile apps use this approach. However, previous studies [27, 44] indicate that today's commodity mobile devices are far from being powerful enough to perform heavy-duty real-time rendering for high-quality VR. Other concerns include excessive energy consumption and heat dissipation.

Server Performing Online Rendering. This design option offloads the rendering task to an (edge) server, which performs real-time rendering of the VR scene for all users based on their positions and viewports. The rendered scenes are then distributed to the users wirelessly as encoded video frames. This approach has been adopted by a prior single-user, cloud-assisted VR system [44]. It drastically reduces the client-side overhead, but in the multi-user scenario, the rendering and video encoding workload becomes too high for a single server to handle. To illustrate this, we perform an H.264 encoding experiment on a high-end workstation equipped with an Nvidia GTX 1080 GPU. The achievable encoding performance is 92 FPS, 199 FPS, and 342 FPS for 4K, 2K, and 1080p resolutions, respectively. This clearly cannot support 10+ users, each requiring a frame rate of higher than 60 FPS.

Server Performing One-time, Exhaustive Offline Rendering. The server exhaustively enumerates all possible views at all positions, renders them at a high quality, encodes them into video frames, and saves the frames in the storage [27]. At runtime, the server simply retrieves and transmits the pre-encoded frames based on each user's position and viewport. In this way, the rendering/encoding overhead at runtime is completely eliminated, so the server can easily scale to tens or even hundreds of simultaneous users. These benefits come at the cost of high storage usage, which is largely not an issue given the cheap storage today.

System Architecture. Firefly employs the third approach given its good runtime performance and superior scalability. Figure 1 plots the overall architecture. As shown, Firefly consists of a content server and multiple commodity mobile

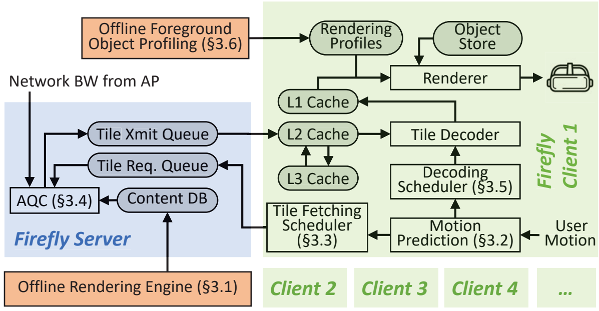


Figure 1: The Firefly system architecture.

devices. They are wirelessly connected through a WiFi access point (AP). This setup can be easily realized in enterprise or home environments at a very low cost. Note that prior work [27] applies offline rendering to a single mobile device for local VR scenes, while Firefly further extends this concept to networked multi-user VR where offline rendering is found to be an indispensable mechanism ensuring the scalability.

The server consists of a content database that stores rendered/encoded content indexed by a user's position and viewing direction. The database is built by the Offline Rendering Engine that performs the aforementioned exhaustive content generation (§3.1). Another critical component is the AQC module that is introduced to scale the system and to handle the wireless bandwidth fluctuation. It determines in real-time the content quality for each user. Designing AQC is challenging due to multiple requirements including boosting users' QoE, maintaining good performance, ensuring scalability, and achieving fairness. We detail its design in §3.4.

On the client side, there are two high-level design choices on the content fetching strategy for background frames. First, the client can prefetch all surrounding frames at every new virtual position [44]. However, this technique may consume high bandwidth with a considerable amount of wasted traffic (*i.e.,* the fetched content is not viewed by the user, see our evaluation in §5.2), making it infeasible for multi-user VR. Second, to reduce the bandwidth footprint, the client can use its historical motion trajectory to predict the future viewport and to prefetch only the portions that will likely be consumed in the near future. Firefly is the first to incorporate this viewport-adaptative approach into generic VR using robust motion prediction (§3.2,§3.3). The client also efficiently manages its local cache (§3.5) and handles foreground dynamic objects in an adaptive and scalable manner (§3.6).

3 System Design

3.1 Offline Rendering Engine

The offline rendering engine produces the content database. The whole VR world is discretized into grids. At each grid position that the user can reach, the rendering engine renders a *mega frame* that captures the 360° panoramic view [28] that the user can possibly perceive at a high quality. Firefly uses Equirectangular projection [7] to generate the panoramic representation, but other projection algorithms [8, 14, 67] can also be applied. As shown in Figure 2, besides the color frame (top), a mega frame also includes a panoramic depth map (bottom) where the brightness of each pixel indicates its distance from the user. The depth map will be used to ensure the correct occlusion when overlaying foreground objects such as avatars of other users onto the scene (§3.6).

We next apply the *tiling* technique [38, 53] by dividing each mega frame into *mega tiles*. Each tile is independently encoded and can be separately transmitted and decoded. The rationale is that, since the user only sees a portion of the whole panoramic scene at a given time, there is oftentimes no need to fetch the entire mega frame. The mega tiles thus allow users to (pre)fetch the content more adaptively at a finer granularity, to reduce the network bandwidth consumption. Note that although viewport-adaptive tiling has been used in 360° video streaming, applying this concept to generic VR (in particular, multi-user VR) is new. Tiling requires the user to predict its viewport, *i.e.,* to determine which tiles to fetch based on the observed viewport trajectory (both translational and rotational), as to be detailed in §3.2 and §3.3.

A decision we need to make is to determine the number of tiles and their layout. While having more tiles provides more bandwidth saving opportunities, in the meantime it increases the decoding overhead and makes compression less efficient. After carefully studying the above tradeoffs using real users' viewport trajectory data (§3.2), we decide to vertically segment each mega frame into four mega tiles as shown in Figure 2. We choose vertical segmentation because according to our data collected from 25 users, users tend to keep their sight vertically centered (*i.e.,* looking at the equator) while moving the viewport horizontally. This makes horizontal segmentation at the equator (0° latitude) inefficient because the vertically centered viewport will always overlap with at least two tiles, *i.e.,* one above and the other below the equator.

As described above, at each position, the offline rendering engine generates four tiles capturing the panoramic view and depth. Each tile is then independently encoded into video frames with multiple quality levels. The rendered and encoded tiles are stored in the content database, indexed by the user's grid (translational) position, the tile ID (rotational position, 1 to 4), and the quality level.

3.2 VR Viewport Movement: Characterization and Prediction

Users' motion makes VR immersive and interactive. In the literature, many studies have investigated users' head *rotational* movement when watching 360° videos [24, 33, 39]. Generic VR differs from 360° videos in that it further involves *translational* movement. To our knowledge, no prior study has comprehensively investigated VR users' motion patterns and their predictability, which are our focus here.

Collecting Viewport Movement Data from Real Users.

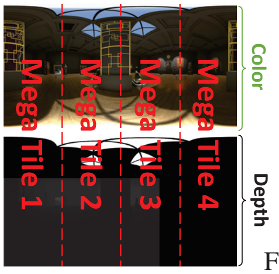


Figure 2: Mega frame.

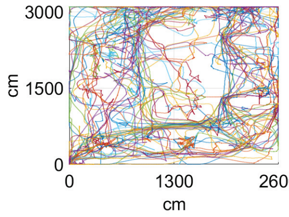


Figure 3: Users' translational trajectories (the Office scene).

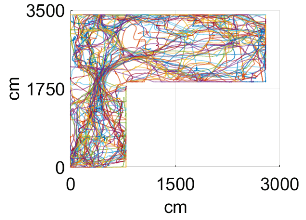


Figure 4: Users' translational trajectories (the Museum scene).

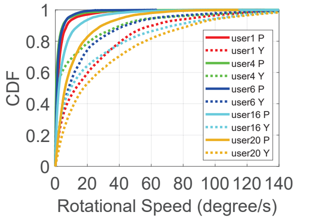


Figure 5: Five users' rotational speed (P=Pitch, Y=Yaw).

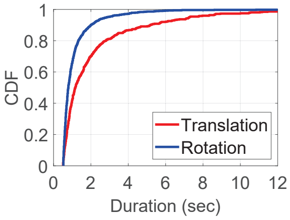


Figure 6: SP duration per pause.

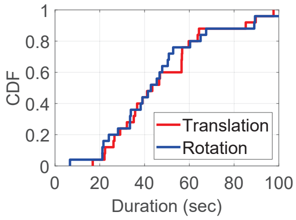


Figure 7: Total SP per user.

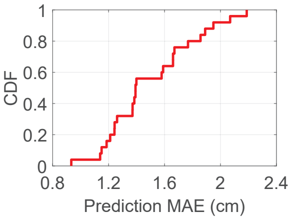


Figure 8: Trans. prediction MAE.

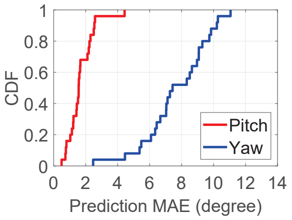


Figure 9: Rot. prediction MAE.

We conduct an IRB-approved user study involving 25 voluntary participants recruited from a large university. Among the 25 users, 9 are female. The users are from 8 departments as undergraduate (16), master (4), and Ph.D. students (5). During the study, each subject wears an Oculus Rift headset [15] connected to a high-end PC. The subject can freely make rotational movement by moving her head as well as perform translational movement using the handheld controller.

We obtain two large VR scenes from the Unity store: Office [16] (30m×26m) and Museum [13] (35m×30m, L-shape). We then develop a custom VR system that loads each scene for the users to explore. Our system logs from each user the precise viewport trajectory. We let each subject explore each scene in a random order for 5 minutes, with an arbitrarily long break allowed between the two sessions.

Motion Trace Characterization. We now characterize the unique dataset above to reveal VR users' motion dynamics and to provide insights for Firefly's design. To begin with, Figures 3 and 4 plot the translational movement trajectories of all users, represented by different colors, for the two VR scenes. As shown, in most locations, the users' trajectories are highly heterogeneous. This finding suggests that the server should not use broadcast or multicast, simply because users typically see different content at a given time.

Fast motion may cause difficulties for viewport prediction. We thus quantify the users' motion speed. The translational movement speed is fixed at 1m/s (set based on reported experiences from another user study) when the user presses the controller button. Figure 5 plots the distributions of rotational movement speed, calculated by sliding a 500ms window over the trajectory, across all window positions for five randomly selected users. As shown, the users exhibit different speeds, whose medians range from 1.3°/s to 18.6°/s for yaw and from 0.5°/s to 7.0°/s for pitch. The median speed across all 25 users is 10.2°/s and 2.4°/s for yaw and pitch, respectively. Interestingly, such speeds match those for typical 360° users [53], implying that translational movement does not necessarily slow down the rotational movement.

Another challenging scenario is users' sudden movement after a stationary period. How often do stationary periods (SPs) occur? Figure 6 plots the distributions of SP duration per pause, which by our definition has to last at least 500ms. Figure 7 plots the total SP duration per user. As shown, an SP is typically short: 69% of translational SPs and 89% of rotational SPs are shorter than 2 seconds. However, Figure 7 indicates that they occur frequently: within a 5-min VR session, a typical user spends 43 seconds (median) being stationary. Such frequent SPs lead to bursty, non-continuous movement patterns that pose difficulties for viewport prediction. To deal with SPs, we design mechanisms such as conservative tile scheduling (§3.3) and bandwidth reservation (§3.4). We also find that translational and rotational SPs are not correlated, *i.e.,* a user is typically looking at a fixed direction while moving, or looking around while standing still. This motivates us to separate the translational and rotational dimensions when performing viewport prediction (see below).

Viewport (Motion) Prediction is required by the tiling scheme (§3.1). We make two decisions regarding Firefly's viewport prediction scheme. First, we decide to run it distributively on client devices to make the server scalable. Second, given the above measurement results, we predict each dimension separately (yaw/pitch for rotational movement and X/Y/Z for translational movement), and then combine them into the final predicted view. We find that this strategy greatly reduces the computational complexity while achieving a decent accuracy – a desirable tradeoff we want to strike. Regarding the actual algorithm, we continuously train a linear regression (LR) model using the motion trajectory observed within a history window of H milliseconds; we then use this model to predict the future trajectory within a prediction window of P milliseconds before discarding the model. The simple LR model is found to be very lightweight yet effective for 360° videos [53]; here we investigate its effectiveness for generic VR motion prediction. Further improvement using more powerful machine learning tools is our on-going work. Ideally, P should be set to the duration of the entire tile processing pipeline (form request being sent to tiles being decoded) plus some safety margin. Guided by this, we set P to 150ms based on empirical profiling. We set H to 50ms based on cross-validating different values of H, which is found to not qualitatively impact the prediction accuracy. Note that when integrated with Firefly, the prediction is performed in an online manner: at runtime, Firefly continuously (1) trains a linear regression model based on the motion trajectory observed within a window (H), (2) uses this model to predict the viewport, and (3) discards this model immediately.

Figure 8 and 9 plot the prediction results for translational and rotational movement, respectively, across all users (H=50ms, P=150ms, the Office scene), with the SPs excluded. The accuracy metric is the mean absolute error (MAE, in distance or degree). The overall accuracy is high: the median MAE is around 1.4 cm for translational movement, and 1.6°/7.4° for vertical (pitch) / horizontal (yaw) rotational movement. The results for the Museum scene are similar. We discuss how Firefly further tolerates prediction errors in §3.3.

3.3 Client-side Tile Fetching Scheduling

The client needs to judiciously decide which (mega) tiles to fetch and in which order. Recall that the client continuously predicts the viewport trajectory within a prediction window (§3.2). The trajectory is a time series of 6-tuples $\{t, x, y, z, pitch, yaw\}$ where t is the (future) timestamp; x, y, and z are the grid position (translational movement); $pitch$, and yaw are the viewing direction (rotational movement). The timestamp difference between two consecutive tuples is $1/F$, where F is the frame rate. In other words, each tuple corresponds to the predicted viewport of a future frame. The client then translates yaw and $pitch$ of each tuple into a list of tiles according to the projection algorithm (*e.g.,* Equirectangular).

The client now has a preliminary list of tiles to be fetched. It next prunes the list using two rules. First, if a tile is already in a client-side cache (§3.5), it will be removed from the list. Second, if a tile appears multiple times in the list, only the earliest appearance (with the smallest t) will be kept. This pruned list where the tiles are ordered by their t values will then be sent to the server. To adapt to users' motion, the above scheduling process is performed continuously on a per-frame basis. The server therefore sees a stream of mega tile lists for each user. We describe how the server processes it in §3.4.

Tolerating Viewport Prediction Errors. Due to users' randomness, viewport prediction errors are inevitable. Firefly employs three mechanisms to tolerate them. First, it uses large tiles (90°×180°) that can absorb rotational prediction errors, as a tile needs to be fetched as long as the predicted viewport has any overlap with it. Second, to further tolerate rotational prediction errors, we virtually enlarge the field-of-view by $p\%$ in each direction when calculating the to-be-fetched tiles. p is configured to 10% given the rotational prediction MAE shown in Figure 9. Third, recall from §3.2 that sudden translational movement after a stationary period (SP) is difficult to predict. To address this issue, when the user is stationary, we add the tiles (corresponding to the current viewing direction) of all four neighboring grids to the predicted tile list. In this way, no matter which direction the user moves towards, the corresponding tiles are always in the to-be-fetched list.

3.4 Adaptive Quality Control (AQC)

AQC takes as input the lists of tiles requested by the users, and outputs each user's appropriate quality level. It runs on the server that has the global knowledge of all users. An ideal AQC algorithm has the following features. (1) For each user, AQC will maximize the quality level while minimizing the stall (rebuffering); meanwhile, the number of quality switches should be minimized to provide a smooth user experience. (2) The selected quality levels should be fair across all the users; in other words, the quality levels should be largely proportional to the users' wireless channel capacities. (3) AQC needs to execute in a fast-paced manner (ideally at the per-frame granularity for each user) to adapt to users' motion. (4) AQC should scale well for multiple users.

At a first glance, AQC is similar to a video bitrate adaptation algorithm where a plethora of studies have been conducted [40,41,51,66]. However, AQC in Firefly is much more challenging. In particular, requirements (2), (3), and (4) do not appear in typical bitrate adaptation algorithms running on a single client for regular video-on-demand services.

In our initial design, we attempt to establish a principled optimization framework that maximizes a QoE (Quality of Experience) utility function. However, we find that this approach is computationally infeasible on a per-tile basis, as the solution space expands exponentially as the number of users increases. To this end, we develop a lightweight, heuristic-based algorithm that produces empirically good quality selection decisions. Our design considers all four requirements mentioned above. It runs efficiently on commodity servers, achieving frame-level scheduling for 10+ users.

AQC Algorithm. We now walk through the detailed logic of the algorithm listed in Figure 10. It uses the available bandwidth obtained from the wireless AP and the recently received to-be-fetched tiles (§3.3) to adjust the quality level (`Q[i]`) for each user `i`. In each invocation, AQC gets the total available downlink bandwidth across all users (Line 01), as well as each individual user's available downlink bandwidth from the AP (Line 03). They represent the global and local network bandwidth constraints respectively (see §4 for their details). λ (empirically set to 90%) adds a safety margin for

```
    n: total number of users
    T: total available bandwidth across all users
    Q: users' current quality levels (input & output)
    Tiles: users' to-be-fetched tile lists (input)
    Q': local copy of Q
    B: individual user's available bandwidth
    λ: bandwidth usage safety margin
    RESERVE: reserved bandwidth for each user
    ---------------------------------------------------------------------
01  T = get_total_bw_from_AP() * λ
02  Q'[1..n] = Q[1..n]
03  B[1..n] = get_individual_bw_from_AP([1..n]) * λ
04  foreach user i:
05      while (bw_util(Tiles[i],Q'[i])≥B[i] and Q'[i] is not lowest):
06          Q'[i] = Q'[i] - 1
07      T = T - min(B[i], max(RESERVE, bw_util(Tiles[i], Q'[i])))
08  if (T < 0):
09      lru_decrease(Q'[1..n]) until (T≥0 or Q'[1..n] are lowest)
10  else:
11      lru_increase(Q'[1..n]) until (T≈0 or Q'[1..n] are highest)
12  Q[1..n] = Q'[1..n]
```

Figure 10: The multi-user AQC algorithm.

tolerating the bandwidth fluctuation. Lines 05–06 deal with the local bandwidth constraint. For a given user **i**'s tiles to be fetched (**Tiles[i]**), as long as their bandwidth utilization (calculated by **bw_util()**) exceeds the available bandwidth **B[i]**, the quality is lowered to avoid stalls. Line 07 then subtracts the user's used/reserved bandwidth from the global bandwidth budget **T**. An important design decision we make is to reserve a certain amount of bandwidth (**RESERVE**) for each user to handle the user's sudden movement (§3.2) that may incur unexpected bandwidth utilization. The reserved bandwidth for each user is set to $\eta T/n$ where T is the AP's total bandwidth, n is the number of users, and η is a tunable parameter. A large η reserves more bandwidth, which can help increase the resilience to users' bursty movement at the cost of a lower flexible (*i.e.,* non-reserved) bandwidth of other users. We empirically choose η=0.75 that yields a satisfactory tradeoff between the two above factors.

We next consider how to estimate the tiles' bandwidth requirement, *i.e.,* realizing **bw_util()** in Line 05. Recall from §3.3 that each tile has its display deadline. Let the total size (in bytes) of the tiles at quality level q with a deadline at or before t_i be $S_{i,q}$. Let t_0 be the current time, t_c be the estimated decoding time, and t_s be the server-side queuing delay. t_0 and t_c are reported by the user and t_s is estimated by the server. In order to not miss the deadline t_i, the required bandwidth should be at least $b(t_i) = S_{i,q}/\max\{0, (t_i - t_0 - t_c - t_s)\}$ (it can be ∞ when a stall occurs). Then the overall required bandwidth is conservatively estimated as $\max_{t_i}\{b(t_i)\}$.

Lines 08 to 11 deal with the global bandwidth constraint. If the global bandwidth budget **T** is depleted (Line 08), then we reduce the users' quality levels (Line 09); otherwise we try to increase them (Line 11). To facilitate fairness and make the quality switch smooth, the decrease/increase of the quality levels is performed in a "least recently used (LRU)" manner, one user at a time, *i.e.,* the user whose quality level was least recently changed is selected. The quality level increase is subject to the local bandwidth constraint.

Since users' requests arrive asynchronously, AQC needs to be invoked to update **Q[1..n]** whenever a new request arrives. Then the tile transmission thread will retrieve the tiles from the content database and put them into the tile transmission queues. If the requested tiles for a user change, or if AQC produces a different schedule in a future invocation for this user, the not-yet-transmitted tiles in the user's queue will be updated. Thanks to AQC's lightweight nature, users' motion and network bandwidth fluctuation will be immediately reflected in the tiles' quality levels, making Firefly robust.

3.5 Client-Side Hierarchical Cache

When a user receives mega tiles from the server, the tiles will be cached, decoded, and rendered. Since tile decoding takes non-negligible time, it needs to be performed in advance. Firefly, a *decoding scheduler* determines which tiles to decode. Its logic is similar to the tile fetching scheduler (§3.3), by using the viewport prediction results. Predicted tiles with a closer display deadline take a higher decoding priority.

To handle large VR scenes, Firefly needs to fetch and decode a large number of tiles. Firefly thus employs a 3-layer hierarchical tile cache. Borrowing the CPU cache terminologies, we name the three layers L1, L2, and L3. Residing in the GPU memory, The L1 cache stores *decoded* mega tiles that can be immediately rendered by the GPU. It is the fastest cache, but its capacity is the smallest (hundreds of tiles) due to the large size of decoded tiles and limited GPU memory. The L2 cache stores *encoded* tiles in the main memory with a capacity of thousands of tiles. The L3 cache dumps encoded tiles in the persistent storage; it has the largest size but is the slowest. When a tile arrives, it is first stored in L2 cache; if L2 is full, some old tiles in L2 may be swapped to L3 in an LRU manner. The L2-to-L3 swap involving writing to flash drive, and is thus performed in a batched manner for good write performance. Swapping back from L3 to L2 is triggered by the decoding schedulers' decisions. This typically occurs when a user visits a previously explored grid position.

3.6 Handling Dynamic Foreground Objects

A VR scene may consist of a background view as well as one or more foreground objects. The background view at a specific virtual location is static. Due to its large area and complexity, its rendering typically dominates the workload for preparing the scene. In contrast, foreground objects are more dynamic and less complex than a background scene. Their examples include moving objects (*e.g.,* other users' avatars) and interactive objects (*e.g.,* a virtual control panel). Despite being less complex than the background view, due to their dynamic and interactive nature, failure to render foreground objects in time may also cause considerable QoE degradation.

Firefly employs two mechanisms to handle foreground objects. First, objects' 3D models (polygons, textures, *etc.*) are distributed to the clients *offline*. This reduces the network bandwidth consumption and eliminates the server's rendering workload at runtime. In a typical VR scene, the objects' 3D

Quality	High	Medium	Low
# Polygons, Size (MB)	30016, 3.30	14566, 1.30	7283, 0.68

Table 1: Three quality levels of the avatar object.

Client Device	High	Medium	Low
Samsung Galaxy S8 (SGS8)	✗,✗,✗	✓,✓,✗	✓,✓,✓
Samsung Galaxy S10 (SGS10)	✓,✓,✓	✓,✓,✓	✓,✓,✓
Samsung Galaxy Note 8 (SGN8)	✓,✗,✗	✓,✓,✗	✓,✓,✓
Motorola Moto Z3 (Z3)	✓,✗,✗	✓,✓,✓	✓,✓,✓

Table 2: Rendering profiles for different phones: whether 60+ FPS can be achieved with 3,6,9 concurrent objects in different qualities.

models are not large (*e.g.,* tens of MBs in total) so they can be bundled with the app installation package or be fetched when the app launches for the first time.

Second, foreground objects are rendered locally by the client. This eliminates the uncertainty caused by the network as well as the potential resource competition from other users compared to a server-side approach. A challenge here is that the number of objects appearing in the viewport may change dynamically. If there are too many objects, the local rendering may still become the bottleneck. For example, in multi-user social VR, a user "sees" other users as 3D avatars; depending on the users' position, more than 10 avatars may appear in the viewport, incurring high rendering overhead. To address this challenge, Firefly supports trading off the rendering quality for a high frame rate. Specifically, the client creates low-quality versions for each object type by downsampling its polygon meshes. Table 1 shows an example of an avatar object originally with 30K polygons. Firefly downsamples them (using Blender [4]) to the medium and low quality with 14.6K and 7.3K polygons respectively. This downsampling process is an offline, one-time effort. Then at runtime, depending on the number of objects to be rendered, their qualities are dynamically determined to facilitate 60 FPS. Downsampling may also use more sophisticated polygon simplification techniques such as bounded-error polygon simplification [42], progressive encoding [45], or adaptive display elision based on the size of the object and its position in the scene [34].

To properly determine the objects' qualities, each client creates a *rendering profile* offline. Let us first assume that there is only one object type (*e.g.,* the avatar). As exemplified in Table 2, for each quality level, the profile maps the number of concurrent objects (3, 6, 9 are shown) to whether 60+ FPS can be achieved. Note that we assign the same quality to all objects to simplify the quality selection. The profile is created by the client through automated tests. During a test, the client is also performing tile decoding/rendering to mimic the workload of generating the background view. Then at runtime, the client can directly consult its profile to determine the objects' quality level. For example, when there are 6 objects, SGS8 should use the medium quality (Table 1) to achieve 60 FPS. When there are multiple types of objects, it may be infeasible to exhaustively enumerate their combinations. In this case, we can apply simple machine learning to predict the rendering performance, using features such as the number of objects, the total number of polygons, *etc.* We leave this as future work.

4 System Implementation

Client and Server. We have integrated the components in §3 into the holistic Firefly system that works on commodity Android/Linux OSes. The client is implemented using Android SDK with a total line of code (LoC) of 14,900. Tile decoding is realized using the low-level Android MediaCodec API [3]. We leverage multiple concurrent hardware decoders, whose optimal number depends on the device, to boost the decoding performance. We use OpenGL ES to perform tile projection/rendering, and use the OpenGL FBO (Framebuffer Object) to realize the L1 decoded cache (§3.5). We have successfully tested Firefly on four mobile devices: SGS8, SGS10, Moto Z3, and SGN 8 (full names in Table 2). These devices can be readily plugged into affordable VR headsets. The rotational and translational motion is provided by the on-device motion sensors and the VR headset controller, respectively. The server is implemented on Ubuntu 16.04 with about 1,000 LoC. The clients and the server communicate over TCP.

WiFi AP. The clients and server are wirelessly connected by a commodity WiFi AP. Since the server is only one wireless hop away from the users, AQC can directly obtain accurate global and per-user available bandwidth from the AP (Line 01 and 03 in Figure 10). This avoids the error-prone bandwidth estimation process widely used in Internet video streaming. To obtain the AP-wide overall bandwidth, we modify the AP's firmware to collect statistics on the maximum PHY rates of the clients, the wireless bandwidth used (20–160MHz in 5GHz Wi-Fi bands), and the busy channel time from hardware registers. To estimate each user's available bandwidth, we also collect statistics on the PHY rate and the frame error rate. The available bandwidth for a client i is estimated as $\Phi_i(1-\varepsilon_i)(1-U)O^{TCP}/N$, where Φ_i is its PHY rate, ε_i is the error rate, U is the channel busy airtime, N is the number of clients taking into account that the airtime will be shared fairly among clients, O^{TCP} is the TCP overhead estimated offline through bandwidth saturation experiments. Similar statistics are available on other APs via interfaces such as WebUI.

The Offline Rendering Engine (§3.1) consists of a rendering engine (developed in C# using Unity) and a mega tile encoder (developed in Python) with a total LoC of 1,500. We use H.264 encoding supported by all mainstream mobile devices.

5 Evaluation

5.1 Evaluation Setup

Content Preparation. We use two commercial VR scenes purchased from the Unity store: Office [16] (30m×26m) and Museum [13] (35m×30m, L-shape). The offline rendering engine (§3.1) discretizes both scenes into 5cm×5cm grids, which are fine-grained enough to provide a smooth translational movement experience. The offline engine renders each panoramic frame in 1440p (Quad HD, 2560×1440) resolu-

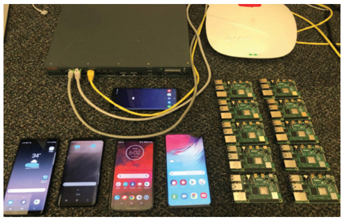


Figure 11: Our equipment: phones, Raspberry Pis, and WiFi AP.

tion, and encodes each mega tile into four quality levels, using the following CRF (Constant Rate Factor) values: 19, 23, 27, 31. A higher CRF corresponds to a lower quality and a lower encoded bitrate. The CRF values are selected by following prior recommendations [17, 20] where the encoded bitrate ratio between two neighboring quality levels is approximately 1:1.5. The content database size is 137 GB and 99 GB for Office and Museum, respectively. When exploring each scene, a user can see other users as avatars, which are rendered by the client as foreground objects. The statistics of the avatar's three quality levels are listed in Table 1.

Hardware and Software. As shown in Figure 11, we use 15 client devices. 5 of them are COTS smartphones with different computational capabilities: SGS8×2 (released in 2017), SGN8 (2017), Z3 (2018), and SGS10 (2019). They all run unmodified Android 9.0. For the remaining clients, we use 10 Raspberry Pi 4 (model B) to emulate them, each having a quad-core ARM Cortex-A72 CPU @ 1.5GHz and 2GB memory. The Pis run Raspbian OS (Debian v10 with Linux kernel 4.19). We run full-fledged Firefly on the 5 smartphones. For the Pis, we create an emulated version of Firefly by replacing the decoding and rendering components with their emulated counterparts. The decoding/rendering latency is properly emulated using the numbers profiled from the 5 smartphones. All other components such as AQC, viewport prediction, tile fetching scheduler, decoding scheduler, L2/L3 caching are identical to those running on a real Firefly client. The server is a desktop PC with an octa-core CPU @ 3.6GHz, 16 GB memory, 1TB disk, and Ubuntu 16.04. The server does not have a dedicated GPU. Clients and server are connected by an Aruba AP running 802.11ac on 80MHz bandwidth.

Physical Environment. The experiments are conducted in a typical office room (7.9m×7.3m) where all the devices, the server, and the AP are located. We distribute the devices at random locations. We find that their locations have a small impact on network performance. For a single device, placing it at the spot nearest to the AP and the spot furthest from the AP yields a throughput difference no more than 11%.

Experimental Approach. To ensure reproducibility, our high-level experimental approach is to replay real users' motion traces collected in §3.2. Recall that we have 25 user traces and 15 devices. In each run, we randomly pick 15 users and assign them in a random order to the devices. Each device then replays the corresponding user's motion trace by feeding the sensor stream to Firefly with precise timing. By default, each experiment consists of 5 back-to-back runs with different user-to-device assignments. We set the users' field-of-view (FoV) to a typical value of 100°×90° [53]. Unless otherwise mentioned, the presented results are based on the Office scene as the results for the Museum scene are qualitatively similar.

5.2 Overall Performance Comparison

We first evaluate the overall performance of Firefly, with the following metrics. (1) **Missed Frame Count (MFC).** In our client implementation, a high-precision rendering timer is triggered every 15ms (or 66.67 Hz). If a frame is not ready at the current timer event, it needs to wait for the next timer event, *i.e.,* after 15ms. In this case the client reports one MFC. MFC is highly correlated with the motion-to-photon delay [69], the time needed for a user's motion to be reflected on the display. When MFC=0 (the ideal case), the motion-to-photon delay is minimized to no longer than 15ms, *i.e.,* the motion is reflected in the immediate next frame. When MFC>0, a stall occurs. (2) **Average frame rate** is measured by sliding a 1-second window over a VR session and calculating the *average* FPS within each window. Our target FPS is 60. (3) **Stall duration** is the rebuffering time experienced by a user. We normalize it to seconds per minute for a VR session. This metric is correlated with the MFC. (4) **Content Quality.** Recall that a tile's quality is defined by the CRF value ∈{19,23,27,31} (§5.1). We then define the quality of a frame as the average quality of all tiles visible in the viewport. (5) **Inter-frame Quality Variation** is measured by sliding a 1-second window over a VR session and calculating the standard deviation of all frames' quality values (defined above) within each window. Since frequent quality switches degrade the QoE [66], a lower value of this metric is preferred. (6) **Intra-frame Quality Variation** of a frame is defined as the standard deviation of the quality values of all tiles appearing in a frame's viewport. Similar to the inter-frame quality variation, we prefer a lower intra-frame quality variation. Metrics (4), (5), and (6) are defined for the background view only. We evaluate the adaptation mechanism for foreground objects in §5.3.

Approaches to Compare. We compare three approaches: (1) full-fledged Firefly, (2) full Firefly with perfect prediction, and (3) the multi-user version of Furion [44]. Approach (2) represents an ideal scenario where users' viewport trajectories are known a priori. It helps us understand how much performance improvement we can further gain by having the perfect knowledge of users' motion. Regarding Approach (3), Furion is the state-of-the-art solution for single-user untethered VR. We create a multi-user version of Furion as follows. We use the full Firefly as the base (to handle multi-user), and then make (and only make) the following modifications according to Furion's design. First, we remove viewport prediction that Furion does not perform. Second, Furion does not use viewport-adaptation; the client instead always requests for all tiles belonging to all four neighboring grids; we thus modify

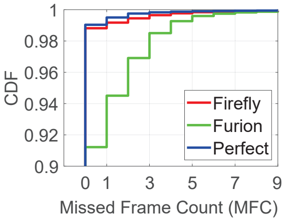


Figure 12: Missed frame count.

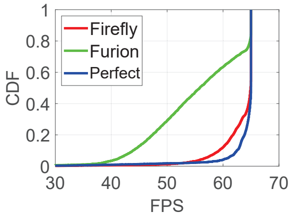


Figure 13: Average FPS.

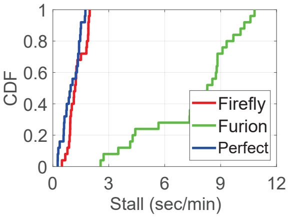


Figure 14: Stall duration.

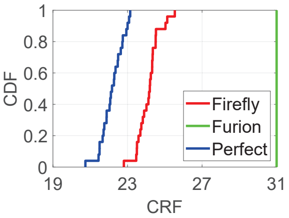


Figure 15: Content quality.

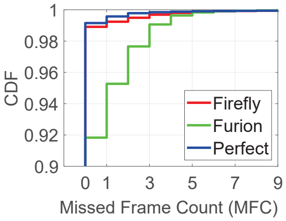


Figure 16: MFC (Museum).

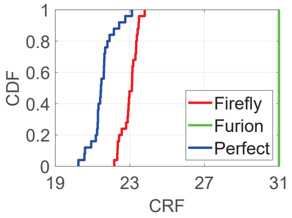


Figure 17: Quality (Museum).

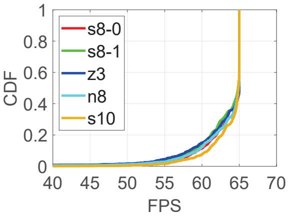


Figure 18: FPS fairness.

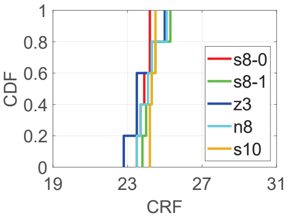


Figure 19: Quality fairness.

the tile scheduling module (§3.3) accordingly.

We next present the results for the Office scene. Figures 12, 13, 14, and 15 plot the distributions of the aforementioned four metrics: MFC (across all timer events), average FPS (across all 1-sec windows' measurements), stall (across all users' sessions), and average content quality (across all users' sessions). Thanks to its adaptiveness to available network/computation resources and its resilience to motion prediction inaccuracy, Firefly achieves overall good performance across all these metrics, which are the same or only slightly worse compared to Firefly with perfect prediction. Specifically, (1) 99% of the timer events (99% for perfect prediction) have MFC=0, *i.e.,* a motion-to-photon delay ≤15ms; (2) for 90%/99% of the 1-sec windows (95%/99% for perfect prediction), the average FPS is at least 60/50 FPS; (3) the median stall duration is only 1.2 sec/min (1.0 sec/min for perfect prediction); (4) the median content quality is around CRF 24.2 (CRF 22.2 for perfect prediction). In Figure 15, the slightly lower quality compared to that of perfect prediction is due to the additionally fetched tiles. The bandwidth consumed by these tiles is wasted because they are not viewed by the users due to viewport prediction errors.

The multi-user Furion exhibits much worse performance. This is because without prediction, it can only blindly fetch an excessive number of tiles without any prioritization. As a result, the bandwidth consumed of many tiles is wasted, leading to a much lower content quality; wasted tiles may also cause head-of-line blocking for useful tiles, causing stalls and a lower FPS. The results for the Museum scene are qualitatively similar, as exemplified in Figures 16 and 17, which plot the MFC and content quality results, respectively.

We also measure the inter/intra-frame quality variations, and find them to be low. For Firefly, the 25th, 50th, and 75th percentiles of the inter-frame quality variation (across all 1-sec windows' measurements) are 0, 0.2, and 0.3, respectively; the 90th percentile of the intra-frame quality variation is 0. Both metrics are very close to Firefly with perfect prediction. The low quality variations are attributed to AQC's quality selection mechanism. It (1) assigns the same quality to all the tiles in a viewport and (2) performs LRU-style quality changes that not only ensure fairness (to be shown next) across users but also facilitate smooth quality switches for a given user.

Fairness. Figures 18 and 19 plot the distributions of FPS and content quality respectively, for five smartphones. Note that although the instantaneous available bandwidth may differ across the devices, in the long run, each device largely gets an equal share of the bandwidth (as verified by us). Also, since each device replays multiple human users' motion traces, this should largely smooth out the impact of motion diversity among the human users. In addition, the devices' computational power heterogeneity is considered by the adaptive object quality selection mechanism (§3.6). Therefore, we expect the distributions to be similar among the devices. This is indeed shown in Figures 18 and 19, confirming that AQC can achieve a decent level of fairness among the devices.

Real Phones vs. Emulated Devices. We observe small performance differences between the two device groups: the 5 real smartphones and the 10 Raspberry Pis. Their average stall duration (across all users' sessions belonging to each group) differs by less than 2%; for both groups, 99% of the timer events have MFC=0; both groups also exhibit very similar FPS distributions; the median content quality is CRF 24.2 and 26.1 for the phone and the Pi group, respectively. This difference is likely attributed to the conservative emulation settings (*e.g.,* decoding latency) used in emulation. Overall, We believe that Firefly is accurately emulated on the Pis.

5.3 Micro Benchmarks

We now present several micro benchmarks to showcase the impact of key design decisions of Firefly.

Impact of AQC. Figure 20 plots the stall duration across all VR sessions with AQC enabled vs. disabled. When AQC is disabled, we consider two extreme cases: always fetching

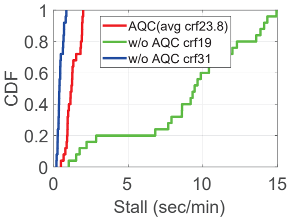


Figure 20: Impact of adaptive quality control (AQC).

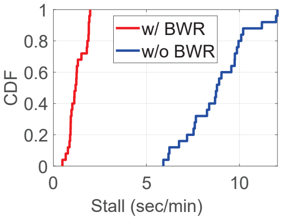


Figure 21: Impact of BW reservation in AQC.

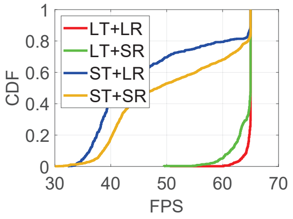


Figure 22: Impact of viewport prediction method.

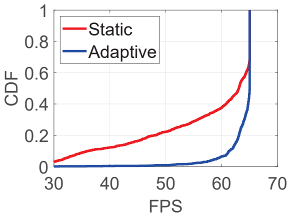


Figure 23: Impact of adaptive foreground object quality selection.

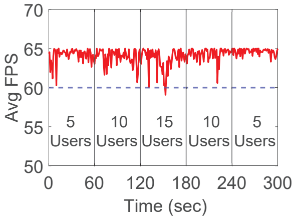


Figure 24: Impact of user dynamics on frame rate.

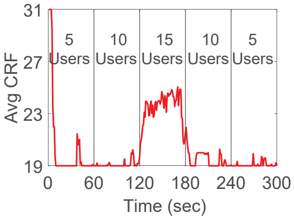


Figure 25: Impact of user dynamics on content quality.

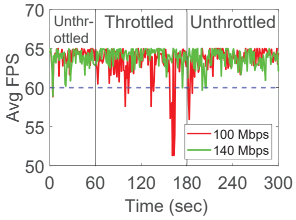


Figure 26: Impact of BW changes on frame rate.

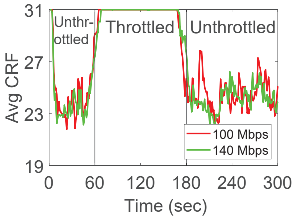


Figure 27: Impact of BW changes on content quality.

the highest quality (CRF=19) and always fetching the lowest quality (CRF=31). As shown, the former suffers from very long stalls (median: 9.6 sec/min); the issue with the latter is the low content quality (CRF 31). AQC instead strikes a much better tradeoff: the achieved average quality is CRF 23.8, while the stall duration is only slightly increased compared to statically using CRF=31 without AQC.

Impact of Bandwidth Reservation in AQC. Recall from §3.4 that we make an important design decision in AQC by reserving for each user a fixed amount of bandwidth to handle the user's sudden motion that may incur unexpected bandwidth utilization. Figure 21 indicates that this mechanism is highly beneficial. If bandwidth reservation (BWR) is disabled, the median stall duration increases drastically from 1.2 sec/min to 8.8 sec/min.

Impact of Viewport Prediction. To justify our viewport prediction design, we consider four variations shown in Figure 22. "LT+LR" is Firefly's approach where we use **L**inear regression (LR) for both the **T**ranslational movement and **R**otational movement prediction; "ST+SR" represents a naïve **S**tatic strategy: directly using the current viewport as the predicted viewport by assuming the user is stationary in both the translational and the rotational dimensions; "LT+SR" corresponds to using LR for translational prediction and Static for rotational prediction; "ST+LR" represents using Static and LR for translational and rotational prediction, respectively. Here, we consider all 25 users' motion traces by replaying them sequentially using one Samsung Galaxy Note 8 phone.

Figure 22 shows that Firefly's approach, LT+LR, achieves the overall highest FPS. Also, LT+SR significantly outperforms ST+LR and ST+SR. This suggests that translational prediction accuracy plays a more important role in determining the system performance compared to rotational prediction accuracy. The reason is that large tiles (90°×180°) can shield many rotational prediction errors (§3.3) but not any translational prediction error.

Impact of Adaptive Object Quality Selection. By analyzing the logs produced by the experiments in §5.2, we find that oftentimes many avatars indeed appear in the viewport: in more than 40% (10%) of the viewports, 4 (8) or more avatars need to be rendered, and this number can reach 10. Too many foreground objects incur high local rendering workload in particular for computationally weak devices. This overhead can be effectively mitigated by the object quality selection scheme (§3.6), which adaptively reduces the fidelity of foreground objects (in our experiments, the users' avatars) to maintain a high FPS. Figure 23 suggests that by disabling this feature (the "Static" curve, which always renders the objects at the highest quality), the FPS drops significantly: the fraction of 1-sec windows with <60 FPS increases from 8% to 37%.

5.4 Adaptiveness to Number of Users

We conduct an experiment to demonstrate that Firefly can properly handle users dynamically joining and leaving the system. We begin with 5 randomly chosen devices at t=0; at t=60s, 5 randomly chosen devices join the system; at t=120s, 5 more devices start their VR sessions; at t=180s, 5 devices leave the system; finally at t=240s, 5 more devices leave. Figures 24 and 25 plot the average FPS and average CRF across all users, respectively, over time. As shown, regardless of the user dynamics, the frame rate almost always stays above 60 FPS. Meanwhile, the content quality well adapts to the bandwidth available to each individual device. When there are no more than 10 devices, each device can enjoy the highest content quality at CRF 19. With 15 devices, AQC reduces the average quality level to ∼24 due to bandwidth scarcity while maintaining fairness across users (Figures 18 and 19). The fluctuations in Figures 24 and 25 (also in Figures 26 and 27 to be described in §5.5) are attributed to our averaging method (first over a 1-second window and then over all users) for calculating each FPS and content quality sample.

5.5 Adaptiveness to Available Bandwidth

We conduct two experiments to investigate how Firefly adapts to changing network bandwidth. In the first one, we begin with unthrottled bandwidth (around 200 Mbps as reported by the AP) at t=0; we then use the Linux `tc` tool [12] to throttle the AP-wide bandwidth to 140 Mbps at t=60s; the bandwidth throttling is removed at t=180s. The second experiment is the same except that the bandwidth throttling is set to 100 Mbps. For both experiments, we fix the number of devices to 15.

Figures 26 and 27 plot the average FPS and average content quality across all users, respectively, over time. When the total bandwidth reduces, the content quality immediately drops to the lowest in order to maintain a high frame rate. For 140Mbps bandwidth throttling, AQC manages to stabilize the frame rate at 60+ FPS. For 100Mbps throttling, each device gets only ~6.7Mbps bandwidth on average that can barely support even the lowest quality level at CRF=31. As a result, the frame rate occasionally drops below 60 FPS.

5.6 Comparison with MUVR

MUVR [47] is a recently proposed, state-of-the-art multi-user mobile VR framework. It is also (to our knowledge) the most relevant work to Firefly. In MUVR, a server maintains a centralized cache that stores the rendered and encoded VR content. Given a user's translational position, the server can directly transmit the view if it is cached; otherwise the server needs to render the view and properly cache it. In their evaluation, the authors emulated 4 concurrent users of MUVR.

We quantify the effectiveness of MUVR on our Office dataset using simulation. The setup is similar to §5.2 where we replay 15 randomly selected users' motion traces 5 times. Meanwhile, we simulate the centralized cache: for every frame, all devices simultaneously "send" their translational positions to the server; upon cache misses, the server will "render" the corresponding positions and add them to the cache. We assume the cache is initially empty and has unlimited capacity. We find that for 27% of the time, there are more than 5 concurrent cache misses, *i.e.,* the server needs to render for more than 5 devices. According to our pilot experiment in §2, this cannot be supported by even a high-end GPU, leading to poor scalability. Firefly eliminates this issue by performing *exhaustive* offline rendering (§3.1). It also introduces other important components that MUVR does not have such as AQC, viewport adaptation, and handling foreground objects.

5.7 Resource Usage and Thermal Overhead

CPU, GPU, and Memory. Firefly incurs acceptable runtime overhead and resource footprint on mobile devices. During a VR session, the CPU usage (reported by the Android Studio Profiler) is no higher than 30% across the five smartphones.[1] The overall memory usage (CPU+GPU) is no higher than 1.6 GB, which is mostly spent on L1 and L2 cache. Note that the cache capacities (L1, L2, and L3) are adjustable in Firefly.

Energy Usage and Thermal Characteristics. To profile the energy usage, we fully charge the five phones, and then repeat the experiment in §5.2 by running on each phone five back-to-back VR sessions. After that (25 minutes later), we record the remaining battery percentage, which ranges from 92% to 96% (average 93.8%) depending on the device's power consumption and battery capacity. We also monitor the CPU/GPU temperature. After continuously playing the VR content for 25 minutes, the highest temperature (either GPU or CPU) among the devices is 50°C, which only feels moderately warm. Overall we think the above energy and thermal characteristics are completely acceptable for mobile VR.

6 Related Work

360° Video Streaming. There exist a plethora of work on streaming 360° videos. Prior systems such as Flare [53], Rubiks [38], Freedom [60], and POI360 [62] also take a viewport-adaptive streaming approach. Some other studies focus on viewport prediction for 360° videos [24, 33, 39]. Compared to the work above, Firefly extends the viewport-adaptation idea to generic VR that involves both the rotational and translational viewport movement. In particular, we demonstrate how viewport adaptation can benefit multi-user VR systems.

Single-user Mobile VR has also been well investigated. Flashback [27] and Furion [44] demonstrate high-quality single-user VR on COTS smartphones. Flashback is a completely local system (on a single device, content stored in SD card). We leverage its core concept of offline rendering to support high-quality, networked multi-user VR. We extend Furion to a multi-user version and quantitatively compare it with Firefly in §5.2. MoVR [22, 23] employs 60 GHz mmWave wireless for mobile VR. Liu *et al.* [48] proposed system-level optimizations for the mobile VR rendering pipeline. Tan *et al.* explored supporting mobile VR over LTE [61]. None of the above work explicitly focuses on the multi-user scenario.

Multi-user VR/AR. Despite a plethora of work on single-user VR, much fewer studies have been conducted on its multi-user counterpart. The most relevant work to Firefly is MUVR [47] that is described in detail in §5.6. Bo *et al.* developed a multi-user 360° video streaming system based on multicast [25]. A recent positioning paper [49] discusses several practical issues of designing a multi-user VR system (without implementation). Some studies investigated multi-user or collaborative augmented reality (AR) [54, 55, 68]. Compared to the above work, Firefly is a generic multi-user VR system. It achieves much better scalability compared to MUVR.

[1] Android Studio Profiler does not report the GPU utilization.

7 Discussion

Efficient Offline Rendering. Future VR applications can involve large and complex scenes, drastically increasing the overheads of offline rendering. Firefly plans to explore well-known rendering optimization techniques [29, 50] which use different hierarchical structures for adapting to the surface tessellation and level of detail.

Handling Dynamic Background. While significantly boosting scalability, Firefly's offline rendering assumes the background content is static (but can be arbitrarily complex). Simple dynamic content involving short animation sequences can still be rendered offline. Complex dynamic content (involving lighting, reflection, *etc.*) has to be rendered at runtime.

Enhancing Firefly using Computer Graphics and Multimedia Techniques. While the contributions of Firefly are mostly on the system side, we are aware that Firefly can be enhanced by various techniques developed from the computer graphics and multimedia community. For example, the 3D models of foreground objects can be simplified using techniques proposed by [31, 32, 35, 56]; visibility or distance culling [34, 36, 52] can be applied to reduce the runtime rendering overhead while maintaining objects' visual qualities; more sophisticated partitioning [58, 59] can be employed to make caching more efficient for both static background and dynamic foreground; more efficient video codec such as H.265 [37] and the next-generation H.266 standard [1] can be leveraged to further reduce the bandwidth footprint for background content delivery; powered by recent advances in deep learning, deep neural networks (super-resolution) can be applied to enhance the image quality [30, 65]. The above approaches are orthogonal to Firefly's exhaustive rendering paradigm and are compatible with the AQC scheme.

Improving Motion Prediction. Firefly employs online linear regression for motion prediction. Despite being simple, it is experimentally demonstrated to be efficient and effective. The prediction accuracy could be further improved using more sophisticated prediction methods. For instance, over 3-DoF (degree-of-freedom) head movement data, deep learning approaches such as LSTM (Long Short-Term Memory) was found to outperform classic machine learning in particular when the prediction window is long [64]. Another promising direction is to enrich the feature set using, for example, velocity, acceleration, and even VR content features such as saliency [33]. We plan to explore the above directions in our future work.

7.1 Lessons Learned

We learned several important lessons from Firefly, which may guide the design of future multimedia systems.

First, Wirth's law [21] also applies to multimedia: the content resolution/quality increase may outpace the graphics technology evolution. While 3D computer games already use some pre-computation techniques such as projecting pre-rendered 2D panoramic background [2] and rendering faraway 3D objects as 2D sprites, we believe that more extensive offline computation and caching will remain a core technique that can scale up high-quality content rendering on commodity hardware, in particular in emerging multimedia services such as mixed reality and cloud gaming.

Second, scheduling content delivery in a multi-user system requires considering a wide range of factors: network bandwidth, device rendering capability, users' QoE, users' interaction, and cross-user fairness. Our experience of developing AQC indicates that while establishing a full-fledged optimization framework may be difficult, a robust heuristic-driven algorithm can work well in practice. In addition, to adapt to users' fast-paced, bursty interactions, the scheduling algorithm needs to run at a frequency that is much higher than traditional videos' bitrate adaptation algorithms [40, 41, 51, 66].

Third, from traditional videos (0 DoF) to 360° videos (3-DoF) and then to VR/volumetric (6-DoF), multimedia content tend to become more immersive and interactive. To embrace such trends, future multimedia systems need more intelligence, which is not limited to motion prediction as showcased in Firefly. Elements such as users' eye movement [43, 46], users' voice, salient visual content [33], and sound source, to name a few, can all be leveraged to infer viewers' intention and henceforth to facilitate system-level decision making such as content prefetching and scheduling.

Fourth, in addition to content, client devices, and server, the network (in particular, the wireless one) is also a key component whose interplay with the multimedia system needs to be carefully optimized. The lower-layer wireless channel information could be leveraged to guide network resource allocation. In Firefly, we demonstrate this over 802.11ac WiFi (§4). Similar cross-layer design could be conducted for other WiFi standards (802.11ax [26]) and cellular networks [62, 63].

8 Concluding Remarks

We have demonstrated with Firefly that it is feasible to support 15 VR users at 60 FPS using COTS smartphones and a single AP/server. Our design makes judicious decisions on (1) partitioning the workload (offline vs. runtime, client vs. server), (2) making the system adaptive to the available network/computation resources, both collectively and locally to each user, and (3) handling users' fast-paced, bursty motion. We believe that the core concepts of Firefly are applicable to other multi-user scenarios such as those of augmented reality and mixed reality.

ACKNOWLEDGEMENTS

We thank the voluntary users who participated in our study, the anonymous reviewers for their valuable comments, and Philip Levis for shepherding the paper. This work was supported in part by NSF Award #1903880 and #1915122.

References

[1] 3 New Codecs Coming in 2020. . https://nofilmschool.com/three-new-codecs-are-coming.

[2] An Adventure in Pre-Rendered Backgrounds. https://justinmeiners.github.io/pre-rendered-backgrounds/.

[3] Android MediaCodec API. https://developer.android.com/reference/android/media/MediaCodec.html.

[4] Blender. https://www.blender.org/.

[5] Bluetooth VR controller. https://www.amazon.com/VR-Bluetooth-Controller-Kasonic-Smartphones/dp/B01E7Z72NQ/.

[6] BYOD Popularity. https://www.forbes.com/sites/larryalton/2017/03/27/how-important-is-a-byod-policy-5-strategies-for-millennials/.

[7] Equirectangular Projection. http://mathworld.wolfram.com/EquirectangularProjection.html.

[8] Google AR and VR: Bringing pixels front and center in VR video. https://blog.google/products/google-ar-vr/bringing-pixels-front-and-center-vr-video/.

[9] Google Cardboard. https://vr.google.com/cardboard/.

[10] How to Build a PC for Virtual Reality. https://www.logicalincrements.com/articles/vrguide.

[11] Keeping the virtual world stable in VR. https://www.qualcomm.com/news/onq/2016/06/29/keeping-virtual-world-stable-vr.

[12] Linux TC. http://man7.org/linux/man-pages/man8/tc.8.html.

[13] Museum Unity Asset. https://assetstore.unity.com/packages/3d/environments/museum-vr-complete-edition-89652.

[14] Next-generation video encoding techniques for 360 video and VR. https://code.fb.com/virtual-reality/next-generation-video-encoding-techniques-for-360-video-and-vr/.

[15] Oculus Rift. https://www.oculus.com/rift-s/.

[16] Office Unity Asset. https://assetstore.unity.com/packages/3d/environments/urban/qa-office-and-security-room-114109.

[17] Per-Title Encode Optimization. https://medium.com/netflix-techblog/per-title-encode-optimization-7e99442b62a2.

[18] Samsung Gear VR. https://www.samsung.com/global/galaxy/gear-vr/.

[19] The very real health dangers of virtual reality. https://www.cnn.com/2017/12/13/health/virtual-reality-vr-dangers-safety/index.html.

[20] What Is Per-Title Encoding? https://bitmovin.com/per-title-encoding/.

[21] Wirth's Law. https://www.techopedia.com/definition/24381/wirths-law.

[22] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Cutting the cord in virtual reality. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 162–168. ACM, 2016.

[23] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, pages 531–544, 2017.

[24] Yanan Bao, Huasen Wu, Tianxiao Zhang, Albara Ah Ramli, and Xin Liu. Shooting a moving target: Motion-prediction-based transmission for 360-degree videos. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1161–1170. IEEE, 2016.

[25] Yanan Bao, Tianxiao Zhang, Amit Pande, Huasen Wu, and Xin Liu. Motion-prediction-based multicast for 360-degree video transmissions. In *2017 14th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9. IEEE, 2017.

[26] Boris Bellalta. Ieee 802.11 ax: High-efficiency wlans. *IEEE Wireless Communications*, 23(1):38–46, 2016.

[27] Kevin Boos, David Chu, and Eduardo Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 291–304. ACM, 2016.

[28] Shenchang Eric Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 29–38, 1995.

[29] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Bdam – batched dynamic adaptive meshes for high performance terrain visualization. In *Computer Graphics*

Forum, volume 22, pages 505–514. Wiley Online Library, 2003.

[30] Mallesham Dasari, Arani Bhattacharya, Santiago Vargas, Pranjal Sahu, Aruna Balasubramanian, and Samir R Das. Streaming 360-degree videos using super-resolution. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020.

[31] Xavier Décoret, Frédo Durand, François X Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *ACM SIGGRAPH 2003 Papers*, pages 689–696. 2003.

[32] Carl Erikson and Dinesh Manocha. Gaps: General and automatic polygonal simplification. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 79–88, 1999.

[33] Ching-Ling Fan, Jean Lee, Wen-Chih Lo, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Fixation Prediction for 360 Video Streaming in Head-Mounted Virtual Reality. In *Proceedings of the Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 67–72. ACM, 2017.

[34] Thomas A Funkhouser and Carlo H Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 247–254, 1993.

[35] Michael Garland and Paul S Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, 1997.

[36] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers*, pages 878–885. 2005.

[37] Dan Grois, Detlev Marpe, Amit Mulayoff, Benaya Itzhaky, and Ofer Hadar. Performance comparison of h. 265/mpeg-hevc, vp9, and h. 264/mpeg-avc encoders. In *2013 Picture Coding Symposium (PCS)*, pages 394–397. IEEE, 2013.

[38] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. Rubiks: Practical 360-degree streaming for smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 482–494. ACM, 2018.

[39] Xueshi Hou, Sujit Dey, Jianzhong Zhang, and Madhukar Budagavi. Predictive View Generation to Enable Mobile 360-degree and VR Experiences. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 20–26. ACM, 2018.

[40] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of SIGCOMM 2014*, pages 187–198. ACM, 2014.

[41] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming With Festive. In *Proceedings of CoNEXT 2012*, pages 97–108. ACM, 2012.

[42] Alan D Kalvin and Russell H Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, 16(3):64–77, 1996.

[43] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. Improving user perceived page load times using gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, pages 545–559, 2017.

[44] Zeqi Lai, Y Charlie Hu, Yong Cui, Linhui Sun, Ningwei Dai, and Hung-Sheng Lee. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. *IEEE Transactions on Mobile Computing*, 2019.

[45] Jiankun Li and C-CJ Kuo. Progressive coding of 3-d graphic models. *Proceedings of the IEEE*, 86(6):1052–1063, 1998.

[46] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 67–82, 2018.

[47] Yong Li and Wei Gao. Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–16. IEEE, 2018.

[48] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 68–80. ACM, 2018.

[49] Xing Liu, Christina Vlachou, Feng Qian, and Kyu-Han Kim. Supporting untethered multi-user vr over enterprise wi-fi. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 25–30, 2019.

[50] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *ACM Siggraph 2004 Papers*, pages 769–776. 2004.

[51] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of SIGCOMM 2017*, pages 197–210. ACM, 2017.

[52] Soraia R. Musse, Christian Babski, Tolga Capin, and Daniel Thalmann. Crowd modelling in collaborative virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 115–123, 1998.

[53] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 99–114. ACM, 2018.

[54] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–95. ACM, 2018.

[55] Xukan Ran, Carter Slocum, Maria Gorlatova, and Jiasi Chen. Sharear: Communication-efficient multi-user mobile augmented reality. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 109–116, 2019.

[56] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. In *Computer Graphics Forum*, volume 15, pages 67–76. Wiley Online Library, 1996.

[57] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 45–50. ACM, 2019.

[58] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. In *Computer Graphics Forum*, volume 15, pages 227–235. Wiley Online Library, 1996.

[59] Jonathan Shade, Dani Lischinski, David H Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 75–82, 1996.

[60] Shu Shi, Varun Gupta, and Rittwik Jana. Freedom: Fast recovery enhanced vr delivery over mobile networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 130–141. ACM, 2019.

[61] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Enabling Mobile VR in LTE Networks: How Close Are We? In *Proceedings of SIGMETRICS 2018*. ACM, 2018.

[62] Xiufeng Xie and Xinyu Zhang. Poi360: Panoramic mobile video telephony over lte cellular networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 336–349. ACM, 2017.

[63] Xiufeng Xie, Xinyu Zhang, Swarun Kumar, and Li Erran Li. pistream: Physical layer informed adaptive video streaming over lte. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 413–425, 2015.

[64] Tan Xu, Bo Han, and Feng Qian. Analyzing viewport prediction under different vr interactions. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 165–171, 2019.

[65] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*, pages 645–661, 2018.

[66] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of SIGCOMM 2015*, pages 325–338. ACM, 2015.

[67] Matt Yu, Haricharan Lakshman, and Bernd Girod. A framework to evaluate omnidirectional video coding schemes. In *Proceedings of the Symposium on Mixed and Augmented Reality (ISMAR) 2015*, pages 31–36. IEEE, 2015.

[68] Wenxiao Zhang, Bo Han, Pan Hui, Vijay Gopalakrishnan, Eric Zavesky, and Feng Qian. Cars: collaborative augmented reality for socialization. In *Proceedings of the 19th International Workshop on Mobile computing Systems & Applications*, pages 25–30. ACM, 2018.

[69] Jingbo Zhao, Robert S Allison, Margarita Vinnikov, and Sion Jennings. Estimating the motion-to-photon latency in head mounted displays. In *2017 IEEE Virtual Reality (VR)*, pages 313–314. IEEE, 2017.