Exploiting Coroutines to Attack the “Killer Nanoseconds”

Christopher Jonathan[†] [*] Umar Farooq Minhas[‡] James Hunter[‡]
Justin Levandoski[‡] Gor Nishanov[‡]

[†]University of Minnesota, [‡]Microsoft
[†]cjonathan@cs.umn.edu, [‡]{ufminhas, jahunter, justinle, gorn}@microsoft.com

ABSTRACT

Database systems use many pointer-based data structures, including hash tables and B+-trees, which require extensive “pointer-chasing.” Each pointer dereference, e.g., during a hash probe or a B+-tree traversal, can result in a CPU cache miss, stalling the CPU. Recent work has shown that CPU stalls due to main memory accesses are a significant source of overhead, even for cache-conscious data structures, and has proposed techniques to reduce this overhead, by hiding memory-stall latency. In this work, we compare and contrast the state-of-the-art approaches to reduce CPU stalls due to cache misses for pointer-intensive data structures. We present an in-depth experimental evaluation and a detailed analysis using four popular data structures: hash table, binary search, Masstree, and Bw-tree. Our focus is on understanding the practicality of using coroutines to improve throughput of such data structures. The implementation, experiments, and analysis presented in this paper promote a deeper understanding of how to exploit coroutines-based approaches to build highly efficient systems.

PVLDB Reference Format:
C. Jonathan, U. F. Minhas, J. Hunter, J. Levandoski, G. Nishanov. Exploiting Coroutines to Attack the “Killer Nanoseconds. *PVLDB*, 11(11): 1702-1714, 2018.
DOI: https://doi.org/10.14778/3236187.3236216

1. INTRODUCTION

Modern main-memory databases [14, 25, 26] store their data completely in low-latency volatile DRAM (or non-volatile RAM). Such a design removes the disk-I/O bottleneck that has plagued traditional database systems for decades. However, main memory still has a higher latency than CPU caches, so main-memory databases suffer from a memory-access bottleneck [9,20]. The data structures employed by many in-memory database systems are pointer-based, *e.g.*, a B+-tree [10, 19, 23] or a hash table. Consider a **get(key)** operation on a B+-tree index used by a modern, in-memory key-value store (KVS). Such an operation dereferences several pointers while traversing the B+-tree from root to leaf, resulting in the well-known issue of *pointer chasing* [16]. Each of these pointer dereferences can stall the CPU, if the data being accessed is not already in the CPU cache. Furthermore, most of a given operation’s CPU instructions and pointer dereferences are dependent on earlier pointer dereferences, and thus cannot be issued in parallel. Consequently, CPU memory-access stalls are a significant performance bottleneck for any system (not just database systems) that relies on pointer-intensive data structures [18, 22]. Our goal is to study an approach that uses coroutines to address this bottleneck.

Recently, many software-based prefetching techniques have been proposed [12, 17, 18, 22] to mitigate CPU stalls due to main-memory accesses. The main idea behind these techniques is simple, yet very powerful. For multiple *independent* instruction streams (e.g., a multi-get operation in an index [22]), one can issue memory prefetches, in parallel—thus exploiting memory-level hardware parallelism. When one instruction stream is about to dereference a memory pointer, it issues a software prefetch of the memory address being pointed to and *context-switches* to the next independent stream, resuming the original operation later. This effectively creates a distance between when a memory address is prefetched and when it is actually dereferenced, making it highly probable that when the memory is accessed, it will be available in a CPU cache.

To implement the above solution, many proposed techniques [12, 17, 18] require an almost complete rewrite of the existing code, for example, by requiring the developer to hand-code a state machine [18]. In many cases, the synchronous version of the code needs to be transformed into an asynchronous version. The resulting code looks very different from the synchronous version of the same code, which is still the preferred choice [11]. Thus, in order to get better performance, these approaches sacrifice code simplicity, understandability, and maintainability.

To address the shortcomings noted above, an even more recent approach to “interleaving” exploits stackless *coroutines* [22]—which are also the focus of this work. A stackless coroutine is a special function that can suspend its execution, returning control to its caller, before it runs to completion. At a later time, after some condition has been met, the caller can resume executing the function from its last suspension point. Stackless coroutines, referred to as coroutines from this point onwards, provide an extremely lightweight mechanism for switching contexts, with a suspend or resume overhead comparable to the overhead of an indirect function call (or return). Further, if a coroutine is inlined, this overhead is zero. Thus, coroutines prove to be a highly efficient mechanism for “interleaving.” Also, with coroutines, developers just need to specify suspension points within their synchronous implementation at places where a cache miss is likely. The compiler automatically generates the code needed to suspend and resume the operation—saving and restoring the state of the operation, respectively. Thus,

[*]Work performed while at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07... $ 10.00.
DOI: https://doi.org/10.14778/3236187.3236216

in effect, coroutines allow the programmer to hide memory latency without extensively rewriting code, as is required by existing techniques [12, 17, 18].

Coroutines are an experimental feature in C++, implemented in the Clang 6.0.0 [1] and 2017 Microsoft Visual Studio (MSVC) [6] compilers. Coroutines are currently a Technical Specification; if they are accepted as part of the upcoming C++ standard [7], they will be available widely. Therefore, it is important to understand their performance characteristics and how they apply to building systems—one of the goals of this work.

We note that one of the key requirements to hide CPU stalls due to memory accesses is the ability to "batch" (or group) multiple operations against a data structure and issue them at once. Above, we presented a "multi-get" operation against a B+-tree, or a hash table, as an example. In [22], the authors present index joins for executing IN-predicate queries in SAP HANA as another example, where the values in the IN-predicate list are synonymous to a multi-get operation against the index. More generally, in multi-tenant cloud environments, where many modern database systems operate, it is possible to "batch" requests from multiple users and exploit "interleaving" to improve performance.

We summarize this paper's contributions as follows. We present an in-depth experimental evaluation of coroutine-based approaches to hide memory access latency. We implement multiple data structures that are commonly used in many systems, using coroutines, and compare their performance with state-of-the-art techniques. Specifically, we perform our evaluation on four different case studies, which can be grouped into two main categories: (1) basic data structures, which include hash-index and binary search, where we build on and expand the analyses presented in [18] and [22]; and (2) complex data structures, as used in the state-of-the-art in-memory databases Masstree [21] and Bw-tree [19], where we believe we are the first to implement and evaluate these approaches. Overall, this paper promotes a deeper understanding of how to exploit coroutines-based approaches to build highly efficient systems.

The rest of the paper is organized as follows: Section 2 provides background on software-based prefetching techniques. Section 3 and Section 4 present an in-depth experimental evaluation of coroutine-based techniques to hide memory stall latencies for simple data structures and complex data structures, respectively. These sections also present our analysis of every approach, both at a micro-architectural and a macro level. Section 5 summarizes our findings and further discusses the practical applicability of coroutines-based approaches. Section 6 concludes the paper.

2. BACKGROUND & RELATED WORK

In-memory database systems commonly use pointer-based data structures, rather than the page-based indirection found in disk-based systems. These indexes are usually in the form of either hash-based indexes—e.g., Redis [8] and RAMCloud [24]—or tree-based indexes—e.g., Bw-tree [19], Masstree [21], and CSB^{+}-tree [23]. Generally, hash tables are better for point lookups, while trees are better for range queries. Previous work [16] has shown that CPU stalls due to main memory accesses are a significant performance bottleneck for pointer-based data structures, on current hardware. As a result, multiple techniques have been proposed, in the last few years, to address this bottleneck [12, 18, 22].

The main idea behind these techniques is to execute N operations on a pointer-based data structure at once (e.g., a multi-get operation on a key-value store). Each operation o_i, where $1 \leq i \leq N$, issues a software *prefetch* of the memory that it is going to access. But before actually accessing that memory (e.g., by dereferencing a pointer), it does a *context-switch* to the next operation o_{i+1}, rather than waiting for the memory it prefetched to arrive at the CPU cache. By the time the process returns to o_i, with high probability, the prefetched memory will be present in CPU caches, and thus o_i can continue its execution, avoiding a CPU stall.

Algorithm 1 *GP*-based Hash Index Probe

```
1: struct state { node; }
2: procedure HASH-PROBE-GP(input[], hash)
3:     init value[input.length]      /* output placeholder */
4:     init state[input.length]      /* state for each probe */
5:     for idx = 0; idx < input.length; idx++ do
6:         state[idx].node = hash.get(input[idx])
7:         prefetch state[idx].node
8:     end for
9:     init num_finished = 0
10:    while num_finished < input.length do
11:        for idx = 0; idx < input.length; idx++ do
12:            if state[idx].node == null then
13:                continue
14:            else if input[idx] == state.node→key then
15:                value[idx] = state.node→value
16:                state[idx] = null
17:                num_finished++
18:            else
19:                state[idx].node = state[idx].node→next
20:                prefetch state[idx].node
21:            end if
22:        end for
23:    end while
24:    return value[]
25: end procedure
```

Throughout the rest of this section, we present state-of-the-art techniques that are tailored around the main idea of "interleaving" different execution (or instruction) streams, to avoid CPU stalls. In particular, we use the example of a hash index probe to show how each approach tackles the problem.

2.1 Group Prefetching

Group Prefetching (GP) [12] is a loop-transformation approach that rearranges N identical operations into M predefined code stages. Then, each operation goes through the same code stage at the same time, with overall execution interleaving between different operations. Once all N operations finish executing the first code stage, all N operations move to the second code stage, and so on until all N operations have executed M code stages.

The main advantage of GP is that the code is written in a "semi- synchronous" manner: GP code follows an operation's synchronous model, except that it executes the operation on N keys in parallel, rather than on one key at a time. As all N operations execute the same code stage before moving to the next one, GP does not need to "remember" the stage that each operation is currently executing. However, GP has two main disadvantages: (1) we need to know the number of pre-defined code stages (M) in advance, which is not always possible, and (2) if one of the N operations terminates early—e.g., because a condition has been met— GP cannot start another operation in its place, because the new operation would be executing a different code stage than the others. This second disadvantage introduces many "no-op" operations into the GP pipeline, hurting overall performance [18] when the workload is skewed.

Algorithm 1 shows pseudocode for the GP approach for probing a hash index. The input to the algorithm is a set of keys to be probed, i.e., N keys, and the hash index. There are a total of 2

Algorithm 2 *AMAC*-based Hash Index Probe

```
 1: struct state { key; node; value; stage; }
 2: struct circular_buffer {
 3:     ...                 /* members */
 4:     function next_state() {...}
 5: }
 6: procedure HASH-PROBE-AMAC($input[]$, $hash$, $group\_size$)
 7:     init result_state[input.length]      /* output placeholder */
 8:     init buff[group_size]                /* circular buffer */
 9:     init num_finished, i, j = 0
10:     while num_finished < input.length do
11:         state = buff.next_state()
12:         if state.stage == 0 then      /* Initialize New Probe */
13:             state.key = input[i++]
14:             state.node = hash.get(state.key)
15:             state.stage = 1
16:             prefetch state.node
17:         else if state.stage == 1 then      /* Access Node */
18:             if state.key == state.node→key then
19:                 state.value = state.node→value
20:                 state.stage = 0
21:                 result_state[j++] = state
22:                 num_finished++
23:             else
24:                 state.node = state.node→next
25:                 prefetch state.node
26:             end if
27:         end if
28:     end while
29:     return result_state[]
30: end procedure
```

code stages in hash index probing. In the first stage (Lines 5–8), the algorithm applies the hash function to get the base node for all N keys and then prefetches them. As noted earlier, the main idea is that when we access the base node at a later time, with high probability it will have been prefetched into the CPU cache. In the second stage (Lines 10–23), the algorithm compares all N keys with the keys of the corresponding prefetched nodes. If a match is found for a key or if there are no more nodes to be fetched, then that operation terminates (early exit). For each unmatched key, the algorithm issues a prefetch for the next node.

The main disadvantage of GP is that when an operation o_i has already found its payload, GP just switches from o_i to the next operation o_{i+1} without initializing a new operation (Lines 12–13). As a result, the number of code stages that GP executes for all N operations is equal to the longest chain of nodes that an operation needs to traverse, regardless of whether the other $N-1$ operations have finished. As we show later in the experimental section, GP works well for data structures which have a regular access pattern, but not for data structures with irregular (or skewed) access patterns.

2.2 Asynchronous Memory Access Chaining

Asynchronous Memory Access Chaining (AMAC) [18] transforms a set of operations into a set of state machines, storing each operation's state in a circular buffer. An operation, right before it stalls—e.g., after prefetching the next memory address—does a context-switch to the next operation. By transforming operations into a set of state machines, AMAC allows different operations to execute different code stages at the same time. This is because the circular buffer stores each operation's current state. Furthermore, once an operation terminates, we can immediately start another operation, without waiting for other operations to terminate. However, the main disadvantage of AMAC is that transforming a synchronous operation into a state machine requires a complete rewrite of the code, and the resulting code does not look anything like the original, "synchronous" version, hence, sacrificing code readability and maintainability.

Algorithm 2 shows pseudocode for the AMAC approach for probing a hash index. The group_size parameter, is the number of concurrently executing instruction streams (or operations). AMAC stores the context of each operation in a circular buffer, of size group_size, where the algorithm loops through every operation in the buffer and executes the relevant stage of each operation. There are 2 main stages in the AMAC version of hash index probe. The first stage (Lines 12–16) initializes a new probe for the next key that the algorithm is going to probe. In this stage, AMAC applies the hash function to the key to find the base node, issues a software prefetch for the base node, and transitions to the next stage. The second stage (Lines 17–27) compares the key with the prefetched node's key. If both keys match, AMAC starts a new probe in the place of the current operation by transitioning to the first stage of the algorithm. Otherwise, it issues a software prefetch for the next node and switches to the next operation in the circular buffer. By doing so, unlike GP, AMAC does not need to wait for every probe operation in the circular buffer to finish before it starts a new probe.

2.3 Coroutines

A key requirement for efficient "interleaving" is that a context-switch must take less time than a memory stall. Otherwise, switching contexts adds more overhead than originally imposed by the memory stalls. This requirement renders many existing multi-threading techniques useless, including light-weight, user-mode threads, known as fibers [4] or stackful coroutines [1].

As shown earlier, GP and AMAC satisfy this requirement by carefully hand-coding highly-efficient code, which sacrifices developer productivity. However, synchronous programming is strongly preferred as it is simpler to understand, hence easier to implement, maintain, and debug [11].The key question becomes: *how can we achieve the high performance of GP and AMAC, while maintaining high developer productivity?* Coroutines, described below, use the compiler to achieve the same high efficiency as AMAC, at a fraction of the development cost.

A *coroutine* [13] is a "resumable function" that can suspend its execution, returning to its caller before it completes. A coroutine can be seen as a generalization of a subroutine: a subroutine is just a coroutine that does not suspend its execution, and that returns to its caller once it completes. When a coroutine suspends its execution, it provides its caller a *coroutine handle*, which the caller can later use to resume the coroutine's execution.

A key feature of coroutines is that, by adding a bit of bookkeeping to coroutines' suspend and resume hooks, the developer no longer needs to put all code inside a single function. A coroutine can be called by another coroutine; and both the callee and the caller can be suspended and resumed at multiple suspension points. This feature allows the developer to add coroutines to existing code quickly and easily, as we show later.

2.3.1 *Using Coroutines in C++*

While coroutines have been around for more than 50 years, they are not yet a standard feature of C++. A coroutines specification for C++ has been published by ISO [2] and is under review to become part of the C++20 standard. As of this writing, coroutines support

[1]Stackful coroutines are up to 93% slower than stackless coroutines on Windows.

Algorithm 3 *Coroutines*-based Hash Index Probe

```
1:  function HASH-PROBE-CORO($key, hash$)
2:      node = hash.get(key)
3:      prefetch node
4:      co_await suspend_always{ }
5:      while node do
6:          if key == node→key then
7:              co_return node→value
8:          else
9:              node = node→next
10:             prefetch node
11:             co_await suspend_always{ }
12:         end if
13:     end while
14:     co_return $null$
15: end function
```

is available as an experimental feature in the Microsoft Visual C++ 2017 (MSVC) and Clang 5.X/6.X (Clang) compilers.

With C++ coroutines enabled, the compiler turns any function that contains any of the keywords **co_yield**, **co_return**, or **co_await** into a coroutine; we use only the latter two keywords in this paper. Keyword **co_return** is roughly functionally equivalent to **return**, and keyword **co_await** will optionally suspend the coroutine, as directed by the object being awaited. This is everything the application developer needs to know to get started with coroutines.

Behind the scenes, the compiler generates code to allocate and manage a coroutine frame for each coroutine call, and provides several hooks around **co_await** and **co_return**, which a library developer can use to keep track of the coroutine's state. (Section 4.1 describes a small library we wrote.) Coroutines are stackless, so they must potentially allocate frames for each coroutine call, which could be slow; in practice, our experiments show that the cost of managing coroutine frames can be made insignificant.

In their current form, C++ coroutines provide only minimal language support. Library developers need to define coroutine types, and each coroutine must return an awaitable—an object that can be **co_await**-ed. The simplest way to use coroutines is to use the C++ standard future library and **co_return** a type **std::future<type>**. Unfortunately, this is also the least efficient way. We experimentally verified that, when using **std::future**, excessive heap allocations significantly degrade performance. However,we expect that ready-to-use coroutine libraries will be available, in the future.

To get better performance, library developers also need to define custom awaitable types. In this paper, we define two different types of awaitables:
- A simple awaitable that returns control to the coroutine's caller when the coroutine is suspended. We use this approach for experiments related to existing work [22].
- A **task<type>** library that supports call chains of tasks, returns control to the root task's caller when the leaf task is suspended, and resumes the entire call chain at the leaf task's suspension point, when the root task is resumed. We use this approach to add coroutines to an existing software project, Masstree.

An awaitable can choose whether to execute immediately or suspend. Because CPUs currently do not indicate whether a given address can be found in cache, we always suspend after prefetching an address. As suggested in [22], future processors could significantly reduce the overhead for addresses already in CPU cache by providing a conditional CPU branch instruction.

2.3.2 Interleaving with Coroutines

A recent research effort combines the advantages of GP (preserving the "semi-synchronous" programming model) and AMAC (decoupled execution) by using coroutines [22]. With coroutines, developers only need to add *suspend* statements to an operation's synchronous implementation. The compiler then generates context-switching code—i.e., it generates efficient code to save the coroutine's state at the suspension point and restore that state when the coroutine is later resumed. (Note that the coroutines defined in [22] do not call other coroutines, which we show is possible, and is needed for most practical use cases.) In other words, while GP and AMAC require the developer to implement a state machine for interleaved operations, coroutines take that significant burden from the developer and hand it to the compiler.

In general, the coroutines approach is similar to AMAC because we interleave multiple coroutines at the same time. However, rather than transforming the operation ourselves into a state machine, with coroutines we just add a suspension point to the synchronous execution of the code after every prefetch. Then, the compiler ensures that the coroutine suspends itself, returning to the caller, which will resume the next suspended coroutine.

Algorithm 3 shows pseudocode for a coroutine-based implementation of a hash index probe. The main idea is similar to AMAC, where we set up a circular buffer to maintain each operation's state. In this case, the state is a coroutine, so we do not need to transform the operation into a state machine or manage its stages manually. Instead, we rely on the compiler to save each operation's state when the coroutine suspends itself. On initializing a new key probe, we initialize a coroutine and store it in the circular buffer. The coroutine has two suspension points (Lines 4 and 10), where the coroutine suspends itself and returns back to the caller. When a coroutine suspends itself, the caller will retrieve the next coroutine from the circular buffer and resume it. When a coroutine finishes its execution—i.e., it has found the matching key, or there are no more nodes to fetch—it returns back to the caller, allowing the caller to replace it with a new coroutine, for the next key probe.

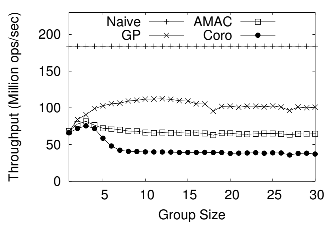
Table 1: Experimental Setup

Processor	**Intel Xeon E5-2690 v4**
# of Sockets	2
# of Cores	28 @ 2.60 Ghz
L1 I/D Cache (per Core)	32 KB/32 KB
L2 Cache (per Core)	256 KB
L3 Cache (per Socket)	35MB
DRAM	256GB
OS	Windows Server 2016 Data Center
Compilers	Microsoft Visual Studio Enterprise 2017 (version 15.4) Clang version 6.0.0

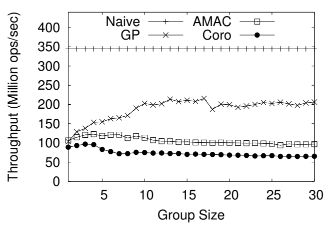
3. SIMPLE DATA STRUCTURES

In this section, we present our experimental evaluation comparing GP-, AMAC-, and Coroutines-based (Coro) approaches to hide memory stall latency for two widely-used, simple data structures: hash table and binary search on a sorted integer array. Our focus is on a *quantitative* comparison using throughput as our main metric. In Section 4, we also comment on *qualitative* aspects such as the impact of different approaches on developer's productivity.

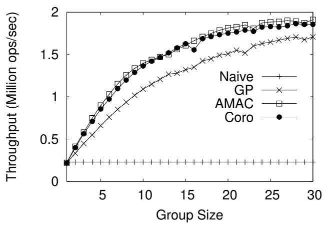
We run all of our experiments on machines running Windows with identical hardware, as shown in Table 1. As coroutines are currently an experimental feature in two C++ compilers—MSVC and Clang—we conduct experiments with both to evaluate compiler differences. For all the experiments reported in this paper, we take an average of at least three runs. We use `_MM_HINT_NTA` for the prefetch instruction, similar to previous work [18, 22]. We ex-



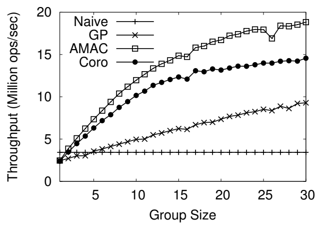
(a) Small Uniform Dataset



(b) Small Zipf Dataset

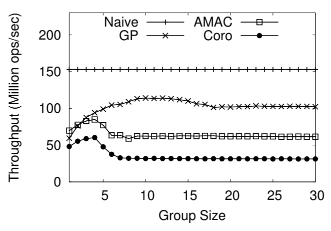


(c) Large Uniform Dataset

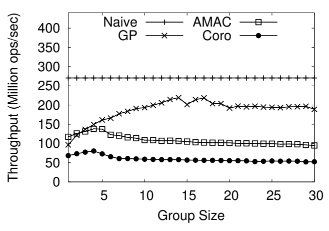


(d) Large Zipf Dataset

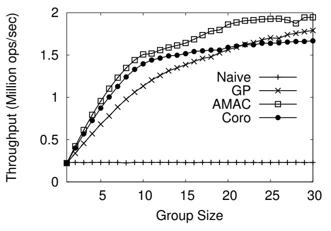
Figure 1: Clang Hash Index Probe 1 Thread Varying Group Size



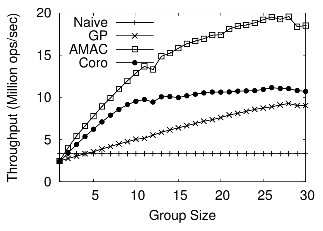
(a) Small Uniform Dataset



(b) Small Zipf Dataset



(c) Large Uniform Dataset



(d) Large Zipf Dataset

Figure 2: MSVC Hash Index Probe 1 Thread Varying Group Size

Table 2: Average Number of Records Per Hash Bucket

Input Size	Uniform	Zipf
0.13M keys (small)	0.83	0.32
1M keys	6.33	2.23
5M keys	31.60	10.45
10M keys (large)	63.21	20.33
50M keys	316.04	95.90

plored other prefetch hints, but found the performance difference to be insignificant on the simple data structures we examine.

3.1 Hash Table

The hash table is a universal data structure. Specifically, in database systems, hash tables are used to implement the popular hash-join and hash-group-by algorithms. Given the importance of hash tables, we start our experimental evaluation by examining the performance of probing a hash table under different prefetch-based (or interleaving) approaches to hide memory latency. For our experiments, we use a canonical implementation of a hash table that uses open hashing: i.e., we use separate chaining (implemented as a linked list) to handle collisions.

We use 4 byte integer keys and 100,000 buckets. Table 2 presents the average size of our collision lists. Note that our hash table does not store duplicate keys, so the number of keys stored in the hash table is less than the input size. For these experiments, the payload size has little effect on performance, since it is fetched only once per operation, i.e., after the probe finds the matching key. We compare a naïve implementation, with no prefetches, to GP, AMAC, and Coro. Our goal is not simply to reproduce the results in [18], but rather to expand the analysis from [18] to an additional interleaving technique, coroutines, and a second compiler, Clang.

3.1.1 Single Threaded, Varying Group Size

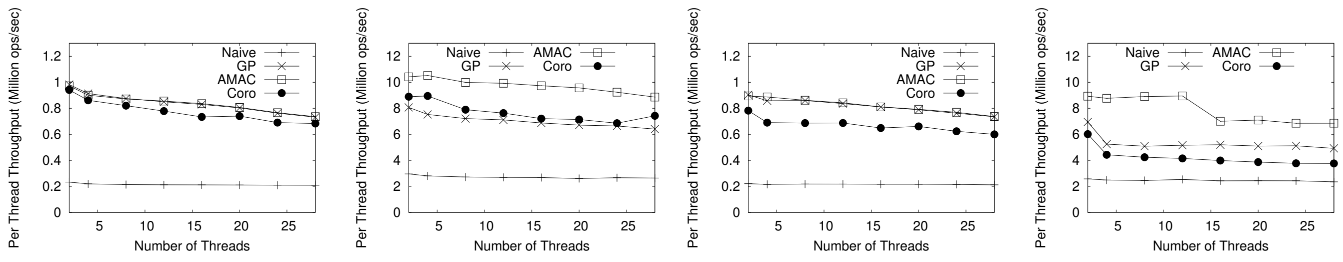
The goal of our first experiment is to understand how group (or batch) size affects the performance of prefetch-based approaches to hash probe. We also study how the distribution of keys used to build and probe the hash table affects performance, to establish how database operators like hash-join and hash-group-by will perform when presented with uniform or skewed workloads—common in practice. To isolate these effects, we run these experiments on a single thread. Figures 1 and 2 present results for the Clang and MSVC compilers, respectively, with group size on the x-axis and throughput, in million operations per second, on the y-axis.

Figure 1(a) shows throughput for Clang using a small data set (0.5 MB) that fits entirely in L3 cache. This experiment uses uniform distribution for the build and probe phases. The results show that, when the hash table fits entirely in the CPU caches, all of the interleaving approaches performed worse than the naïve approach. Naïve is up-to 2.7x 2.9x, and 5.1x faster than GP, AMAC, and Coro, respectively. In this case, the extra instructions for prefetching and interleaving are pure overhead. Of the three interleaving approaches, GP performs the best, because GP has the lowest overhead, in terms of the state that it needs to maintain for interleaving, while AMAC outperforms Coro because Coro's instruction overhead is higher than AMAC's. (Each coroutine has its own coroutine frame, which, with current compilers, requires more bookkeeping than an AMAC state.) The optimal group size is around 3 for AMAC and Coro, and around 12 for GP, roughly the same as in [18].

Figure 2(a) shows results for the same experiment, using MSVC. Naïve throughput drops by 17%, and Coro performance is also lower. This illustrates an important point: when the workload is small enough to avoid memory stalls, the compiler makes a big difference.

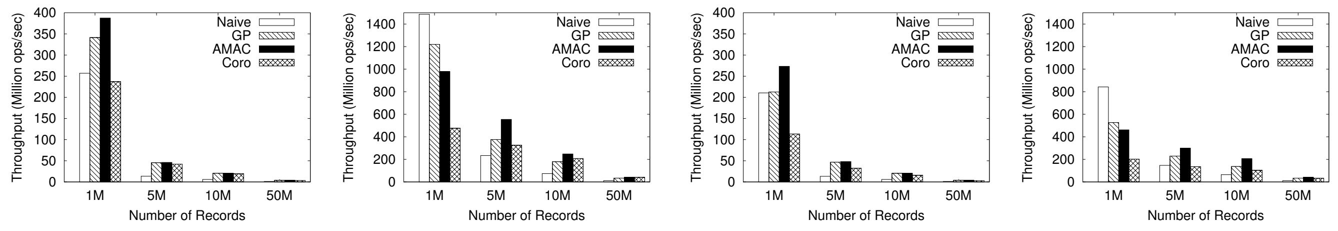
Figures 1(b) and 2(b) show the results with the small dataset, for Clang and MSVC, respectively, but now using Zipfian ("Zipf") distribution to build and probe the hash table, with Zipfian constant = 0.99. A Zipf distribution is skewed, with some keys more likely to be chosen than others, making the CPU caches much more effective; thus, we see an increase in the throughput for all approaches, although the results are otherwise similar to Figures 1(a) and 2(a). Again, MSVC's naïve performance is significantly lower than Clang's.

Figure 1(c) presents the results for uniform distribution with Clang using a hash table with 10 million integers, which without duplicates is approximately 390 MB in memory—much bigger than L3 cache. In this case, all the "interleaving" approaches perform significantly better than the naïve approach. Naïve is slower by up to 7.5x, 8.4x, and 8.2x as compared to GP, AMAC, and Coro, respectively. Figure 2(c) gives the performance with MSVC compiler. From both figures, we can see that the performance is similar to Clang although MSVC's Coro is slower than Clang's.



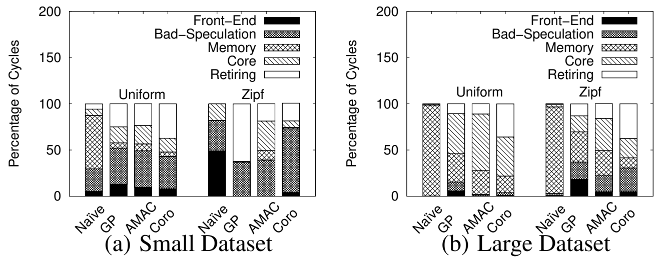
(a) Clang, Uniform (b) Clang, Zipf (c) MSVC, Uniform (d) MSVC, Zipf

Figure 3: Hash Index Probe Varying Number of Threads with 30 Groups, Large Dataset



(a) Clang, Uniform (b) Clang, Zipf (c) MSVC, Uniform (d) MSVC, Zipf

Figure 4: Hash Index Probe Varying Input Size with 30 Groups, 28 Threads



(a) Small Dataset (b) Large Dataset

Figure 5: Hash Probe Microarchitectural Analysis

Although the distribution is uniform, this workload's access pattern is very skewed, since each hash bucket holds an average of 63 keys. Traversing a hash bucket involves a linear search through a linked list, so the number of prefetches per operations varies from 1 to 63. This skew hurts GP's performance, relative to the other two interleaving techniques, as originally reported in [18]. It also causes throughput for all three interleaving approaches to continue to increase with the group size, well beyond a group size of 10.

Inequality 1 from [22] relates optimal group size to computation, memory-stall, and context-switch durations, while Section 5.4.2 of [22] discusses line-fill buffers. The CPUs we tested have 10 line-fill buffers [5], limiting outstanding memory accesses to 10. We note that Inequality 1 does not apply to our experiment, since its access pattern is skewed. Further, we observe that performance continues to increase even after all 10 line-fill buffers are used, because although the hash table and the first few buckets in every chain fit in L3, the remaining buckets do not. This means that some hash probes are satisfied from L3, while others must obtain one or more buckets from DRAM. Figures 1(c) and 2(c) show that maximum interleaving performance is reached when there are > 10 outstanding prefetches, because multiple prefetches from L3 will complete while waiting for one prefetch from DRAM. Using a large dataset with 10 million integers (120 MB in memory), with Zipf distribution, shown in Figures 1(d) and 2(d), yields similar results. The difference is that the skew is now more pronounced, since the CPU caches now hold buckets for the most popular keys.

In Figures 1 and 2, Coro performs significantly worse on MSVC than on Clang. We attribute this difference to the differences in compiler support for generating coroutines code. More specifically, these results show that Clang compiler generates better optimized code for Coro, as compared to the MSVC compiler. We believe MSVC's coroutines specific optimizations are less mature, at the time of this writing, and are expected to improve.

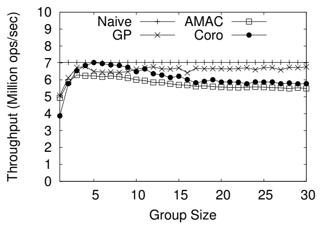
To pinpoint the source of overhead for naïve, we show a micro-architectural analysis using Intel VTune for MSVC in Figure 5. We use a methodology similar to that described in [22]. Figure 5 presents percentage of CPU cycles spent on each stage of the instruction pipeline. In particular, "Memory" denotes the percentage of CPU cycles wasted because the CPU was waiting for data to arrive from main memory. For the large dataset, for both uniform and Zipf, naïve is bounded by main memory accesses. For GP, AMAC, and Coro, CPU stalls due to main memory accesses were significantly less, showing the effectiveness of these approaches.

In summary, this experiment shows that GP, AMAC, and Coro improve hash probe throughput, under uniform and Zipf distributions, due to reduced CPU stalls (Figure 5). The GP approach suffers when using Zipf distribution due to the irregular access pattern. And finally, the Clang compiler generates more efficient code for the Coro approach. We see this consistently in all the results presented throughout the paper.

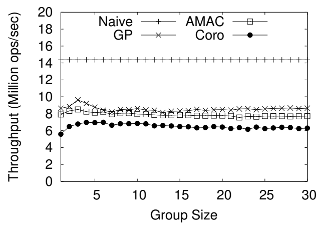
3.1.2 Scalability with Number of Threads

In this next experiment, we want to study the scalability properties of these different approaches for hash probes. Based on the results of the previous experiment, we chose a group size of 30, per thread, for all the prefetch-based approaches. We chose this group size not because it is optimal, but because Figures 1 and 2 show that performance is insensitive to group size around 30. The optimal group size depends on the workload, which is generally not known in advance. We varied the number of threads from 2 to 28 using up-to 14 physical cores, with two hardware threads per core. We used the large dataset from the previous experiment with uniform and Zipf distributions.

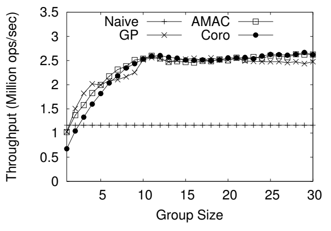
We present the results in Figure 3. For all these figures, we present the number of threads on the x-axis, and the per-thread throughput on the y-axis. These figures show that performance scales almost linearly with the number of threads, for naïve and the three interleaving approaches, although MSVC, Zipf shows a drop in AMAC performance between 14 and 16 hardware threads. This



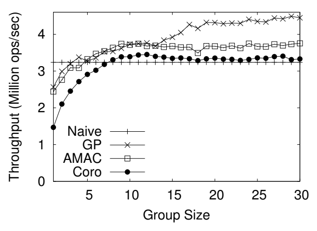
(a) Small Uniform Dataset



(b) Small Zipf Dataset

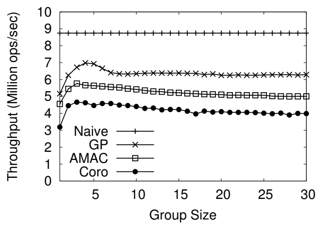


(c) Large Uniform Dataset

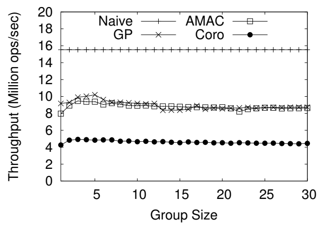


(d) Large Zipf Dataset

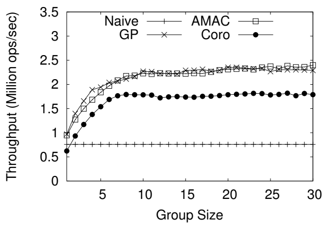
Figure 6: Clang Binary Search 1 Thread Varying Group Size



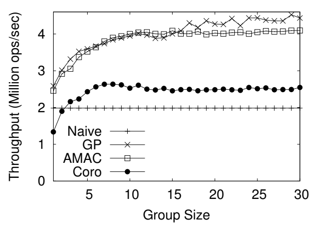
(a) Small Uniform Dataset



(b) Small Zipf Dataset



(c) Large Uniform Dataset



(d) Large Zipf Dataset

Figure 7: MSVC Binary Search 1 Thread Varying Group Size

means that the advantages of interleaving apply even when using more than a single thread. For both Clang and MSVC, all three interleaving approaches maintain a significant advantage over naïve.

3.1.3 Varying Input Size

The goal of our next experiment is to study the impact of data size on throughput of various approaches. We varied the size of the hash table from 1 million to 50 million records, and used uniform and Zipf distribution to build and probe the hash index. We fixed the number of threads at 28, with 30 groups per thread. Figure 4 shows our results, with the number of records on the x-axis, and total throughput on the y-axis.

Figure 4(a) presents the results using Clang for uniform distribution. At 1 million keys, naïve has fully exhausted L1D, L2, and L3 caches; so prefetching is sometimes profitable. This is why AMAC and GP are faster than naïve by about 1.5x and 1.3x, respectively. Coro performs similarly to naïve, because its context-switching overhead outweighs its prefetching benefits, since most probes can be fulfilled by L3 cache. At 5 million keys and higher, all three interleaving approaches significantly outperform the naïve approach since most probes go to DRAM. Figure 4(c) is similar.

Using a skewed distribution, as shown in Figures 4(b) and 4(d), makes prefetching less profitable, since popular keys are already in the CPU caches. And similar to the case when all the data fits in the CPU caches, we pay the overhead of interleaving without getting any benefit. For example, Coro with the MSVC compiler, at 5M records, Zipf distribution, is slower than the naïve approach, while the other two interleaving approaches are faster.

In summary, this experiment shows that the advantages of hiding memory stall latency during hash probe hold over a range of data sizes. At smaller scales, interleaving approaches represent pure-overhead, but as the data sizes grow out of the CPU caches, they provide a significant performance boost over the naïve approach. When using the Clang compiler, AMAC and Coro continue to be the most performant. This shows that, by using Coro, we can reap most of the performance benefits of the AMAC approach while maintaining very high developer productivity.

3.2 Binary Search

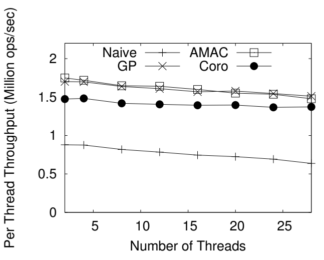
We now present our results with different approaches for binary search over a sorted integer array (4-byte keys), without duplicates. For all the experiments, we fill the sorted arrays with sequentially-generated keys. We present results using both uniform and Zipf distributions to generate search keys. Our experimental methodology follows the same pattern as for the hash probe results presented in the previous section. We use an integer array of size 0.5 MB and 1 GB for the small and large dataset, respectively. We used a hybrid, branching binary search implementation, where the loop body contains an equality test, allowing for early exit. That test almost always evaluates to false, making the branch highly predictable. We intend the comparison ($<$ vs. $>$) to compile into a conditional move, as in [22].

3.2.1 Single Threaded, Varying Group Size

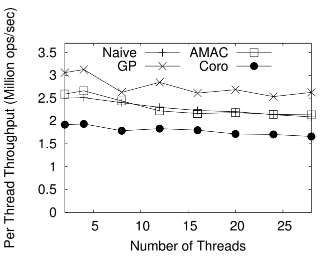
To study the impact of varying group size on binary search throughput, we conduct an experiment with varying group size using a single thread. Figures 6 and 7 present the results of these experiments compiled with Clang and MSVC compilers, respectively. In all of these figures, we present the group size on the x-axis and the throughput, in million operations per second, on the y-axis.

Consider Figure 6(a), which shows the throughput of binary search using the Clang compiler with a small uniformly distributed data set. This data set is small enough (0.5 MB) that it fits entirely in CPU caches. When the integer array fits entirely in the CPU caches, for all group sizes, the interleaving approaches perform worse than the naïve approach, because the cost of interleaving outweighs the benefits of prefetching memory from L3 cache. Naïve is up to 1.4x, 1.4x, and 1.8x faster than GP, AMAC, and Coro, respectively. Similar to hash probe, overall, GP performs the best among the interleaving approaches, since it has the lowest overhead as mentioned in [22]. Figure 7(a) shows the result of this experiment with MSVC. In this case, naïve is up to 1.7x, 1.9x, and 1.9x faster than GP, AMAC, and Coro, respectively. Another notable difference is that Coro now performs worse than AMAC due to inefficiencies in the MSVC compiler.

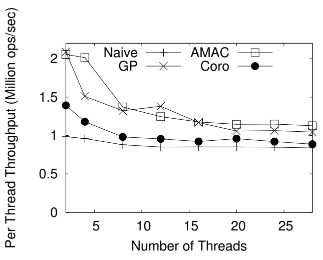
Even more notable, the naïve approach compiled by MSVC is much faster than compiled by Clang for small uniform and Zipf datasets, but much slower for large uniform and Zipf datasets. The MSVC compiler generates a highly-predictable branch and a conditional move, as we intended, but the Clang compiler generates a second, unpredictable branch, instead. For binary search, the choice of compiler can make a big difference. While naïve and



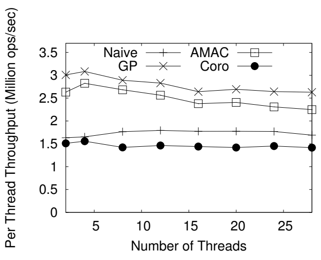
(a) Clang, Uniform



(b) Clang, Zipf

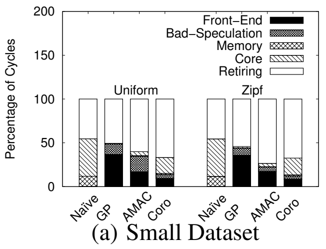


(c) MSVC, Uniform

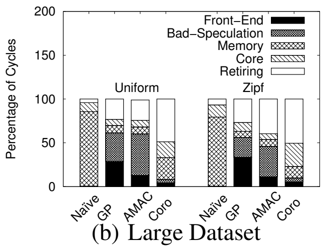


(d) MSVC, Zipf

Figure 8: Large Binary Search Varying Number of Threads with 20 Groups, Large Dataset



(a) Small Dataset



(b) Large Dataset

Figure 9: Binary Search Microarchitectural Analysis

Coro performance vary between the two compilers, AMAC performance does not—the two compilers generate similar AMAC code.

Figures 6(b) and 7(b) show results with the small dataset but now using Zipf distribution for generating the search keys. Since a Zipf distribution means that search keys are likely to be in L1D or L2 cache, the relative advantage of prefetching is much lower than for a uniform distribution, causing the interleaving approaches to perform much worse than the naïve approach.

Now moving onto a 1 GB dataset, which is much larger than L3 cache, Figure 6(c) and 7(c) show that all the interleaving approaches perform significantly better than the naïve approach, when using a uniform distribution. Also, the optimal group size is around 8–12 groups, roughly confirming the results in [22].

When using a large dataset with Zipf distribution with Clang, shown in Figures 6(d) and 7(d), Clang naïve performance is within 25% of Clang's Coro and AMAC, while MSVC naïve performance is around half of MSVC's AMAC. As discussed above, MSVC and Clang compile our naïve approach into very different code.

GP shows better performance than the other interleaving techniques, because the access pattern of binary search is not heavily skewed: half of the sorted array's keys are in the binary search tree's leaf level, a quarter are in the next level up, and so on. GP performance increases beyond a group size of 15, because the Zipf distribution means that where a key sits in the memory hierarchy is heavily skewed.

The micro-architectural analysis presented in Figure 9 also confirms that for large data set, naïve is memory bound 85% of the time, while GP, AMAC, and Coro are memory bound for only 7%, 8%, and 25%, respectively, showing their effectiveness.

In summary, this experiment shows that all three approaches to hide memory stall latency improve binary search throughput, when the data does not fit in the CPU caches, under both uniform and Zipf distributions. Also, interestingly, naïve performs really well when the search keys are generated using a Zipf distribution. Using Zipf effectively reduces the working set size, resulting in a more effective utilization of the CPU caches.

3.2.2 *Scalability with Number of Threads*

We next study how different approaches scale for binary search. Based on the results of the previous experiment, we fix a group size of 20 per thread for all approaches. This is not the optimal group size—which varies between 5, for GP on a small, uniform dataset, and 30, for GP on a large, Zipf dataset—but rather a point at which performance is insensitive to changes in group size. We vary the number of threads from 2 to 28 using up to 14 physical cores, with two hardware threads each. We use the large dataset with uniform and Zipf distributions. Figure 8 shows the results, with number of threads on the x-axis, and per-thread throughput on the y-axis.

Focusing on the results presented in Figure 8(a), using Clang with uniform distribution, we see that all the approaches scale nicely with increasing threads. And naïve is up to 2.4x, 2.3x, and 2.2x slower as compared to GP, AMAC, and Coro respectively. When using Zipf distribution, shown in Figure 8(b), naïve performs similar to AMAC while performing roughly 20% faster than Coro. The reason is that the Zipf distribution utilizes CPU caches much more effectively, as noted earlier.

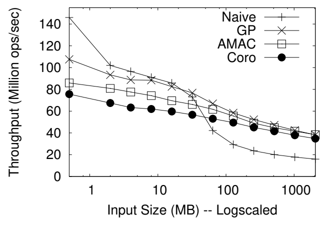
MSVC results for this experiment are presented in Figures 8(c) and 8(d) for uniform and Zipf distribution, respectively. For the uniform dataset, Coro is on average 36% and 28% slower than AMAC, and GP, respectively. For the Zipf case, Coro is on average 10.4%, 46.8%, and 41% slower than naïve, AMAC, and GP, respectively. We believe that this behavior is mainly caused by both: (1) suboptimal support of coroutines in MSVC and (2) effectivenes of CPU caches with Zipf distribution.

In summary, this experiment shows that for Clang, all the approaches scale nicely with the number of threads, and maintain a significant advantage over the naïve case for the uniform case.

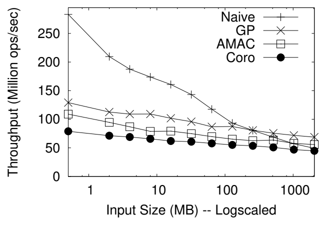
3.2.3 *Varying Input Size*

The goal of our next experiment is to study the impact of data size on binary search throughput. More specifically, we aim to highlight the cross-over point at which the data goes from being able to fit entirely in CPU caches to having to spill to DRAM, and how that impacts throughput. We vary the size of the integer array from 0.5 MB to 2 GB, and use uniform and Zipf distribution. We fix the number of threads to 28, with 20 groups per thread. Figure 10 shows the results, with the size of the sorted integer array on the x-axis (log scale), and the total throughput on the y-axis.

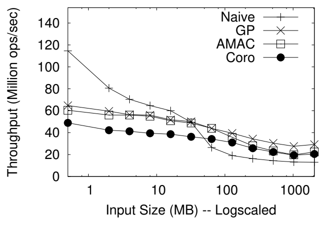
Figure 10(a) presents the results using Clang for uniform distribution. At 0.5 MB, unsurprisingly, naïve is the fastest and is about 1.4x, 1.7x, and 1.9x faster than GP, AMAC, and Coro, respectively. At this point, around half of the array—i.e., all but the leaf level of the binary search tree—can fit in L2 cache. However, we see a performance drop from naïve at 2 MB, where the last two levels of the binary search tree no longer fit in L2 cache. Naïve continues to be slightly faster than the interleaving approaches so long as the dataset fits in L3 cache—which, on our machines, is 35 MB per socket. Therefore, as expected, we see a cross-over point between 32 MB and 64 MB. After the cross-over point, naïve is on average 2.2x, 2.1x, and 1.9x slower than GP, AMAC, and Coro, respectively. Starting at 4 MB, GP continues to outperform all other approaches, before and after the cross-over point. This confirms



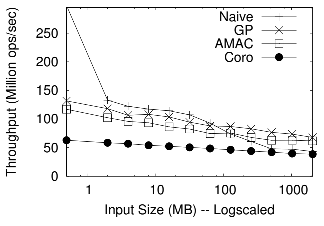
(a) Clang, Uniform



(b) Clang, Zipf



(c) MSVC, Uniform



(d) MSVC, Zipf

Figure 10: Binary Search Varying Input Size with 20 Groups, 28 Threads

the results reported for GP being the best overall approach for binary search in [22]. As noted there, binary search is the best case for GP since all the different "interleaved" executions execute the same code path, which reduces the number of executed instructions. Further, in this case GP's overhead is minimal as the state that needs to be maintained is negligible.

When using a Zipf distribution, as shown in Figure 10(b), at 0.5 MB, naïve outperforms GP, AMAC, and Coro, by 2.2x, 2.6x, and 3.6x, respectively. However in this case, naïve is able to maintain its performance for most of the array sizes where GP and AMAC are only able to outperform naïve when the array size is larger than 256 MB and 1 GB, respectively. Meanwhile, Coro is unable to outperform naïve even with an array size of 2 GB. Again, this is because CPU caches become much more effective with Zipf distribution, thus, benefiting naïve.

Once again, MSVC shows mixed results for this experiment as presented in Figures 10(c) and 10(d) for uniform and Zipf distribution, respectively. With uniform distribution, as expected, we see naïve's throughput drop after the cross-over point. In the case of Zipf distribution, the performance of every approach with MSVC follows our findings with the ones in Clang.

In summary, this experiment highlights the impact of varying data sizes on binary search with different approaches. We see a clear cross-over point from all data fitting in CPU caches to spilling to DRAM. We show how throughput for different approaches is impacted before and after the cross-over point with uniform distribution. At smaller scales, "interleaving" approaches represent pure-overhead, but as the data sizes grow out of the CPU caches, they provide a significant performance boost over the naïve approach. Meanwhile, naïve is able to perform well even when the entire array does not fit in the CPU cache in the case of Zipf distribution. The reason is that most probes will only access certain part of the array, which makes CPU caches become more effective.

4. COMPLEX DATA STRUCTURES

In this section, we show how to modify two widely-used complex data structures, Masstree in Section 4.1 and Bw-tree in Section 4.2, to incorporate coroutines in their complex code-bases. In doing so, our goal is to highlight the qualitative advantages of using a coroutines-based approach in practice. Further, we evaluate the effectiveness of interleaving to hide memory stall latency for Masstree and Bw-tree using the same methodology and experimental setup used in the previous sections.

4.1 Masstree

Masstree is a high-performance, in-memory, key-value store. Masstree splits keys into 8-byte chunks and stores each chunk in a B+-tree. Each B+-tree node is a few CPU cache lines in size. Masstree prefetches each node before processing it, and each node is small enough that Masstree will touch all of the prefetched cache lines when it actually processes the node. (Note that Masstree prefetches using `_MM_HINT_T0`, which prefetches memory into all three levels of the CPU cache, rather than `_MM_HINT_NTA`; this distinction matters to the NUMA results we discuss in Section 4.1.2.)

By prefetching each node, Masstree can effectively execute instructions for free while waiting for the node to be prefetched. The problem is finding useful instructions to execute: the current node cannot be processed until it is available in the CPU cache, and the next node cannot be prefetched until the current node is processed. This problem is not specific to Masstree—it is fundamental to all B+-trees—but it makes Masstree (and any B+-tree) a good candidate for coroutines. Coroutines provide an easy, inexpensive way to get more useful instructions to execute, by interleaving multiple top-level **get()** or **put()** operations on the same hardware thread.

4.1.1 Adding coroutines to Masstree

Masstree is written in C++ and makes extensive use of C++ templates. Masstree implements a top-level **get()** or **put()** operation over several small template functions, with different template parameters, defined inside several header files. Organizing the code in this modular way makes it easy to read and modify, while the C++ compiler will inline these small template functions, generating a fast executable.

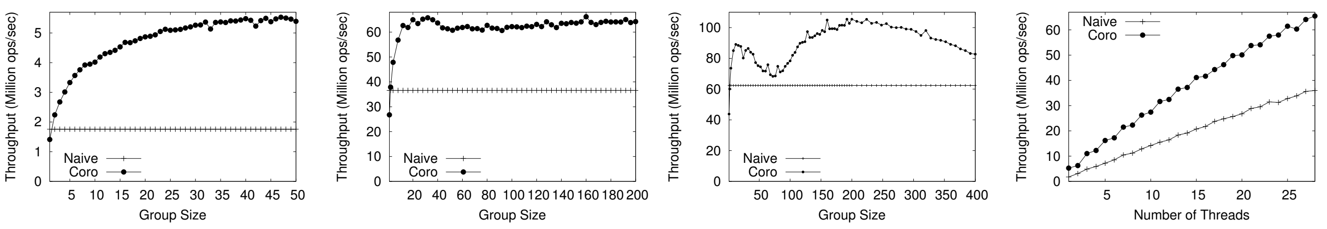
Existing research on coroutines for hiding memory stalls [22] focused on turning a single function into a coroutine, but adding coroutines to Masstree requires either (1) rewriting Masstree so that it uses a single, large function rather than many small, modular, functions; or (2) suspending and resuming an entire call chain of coroutines. The first option is infeasible: it would require substantial developer resources to de-modularize Masstree, and the resulting code would be much harder to understand and maintain.

We chose the second option. To add coroutines to Masstree, we implemented a new **task<type>** class (based on the same principles as [3]) and replaced the default **suspend_always** class. Our **task<type>** class allows a function of return type **T** to be mechanically turned into a coroutine of return type **task<T>**, where each coroutine can **co_await** calls to other **task<type>** coroutines. When the leaf task in a call chain is suspended, control returns to the root task's caller—in our case, a benchmark driver executing a simple round-robin scheduler. Then the driver can do other work, such as resuming another coroutine.

Eventually, the driver will resume the suspended root task, which will actually resume the leaf task at its suspension point. The leaf task maintains a pointer to its caller, so when it exits (using **co_return**), its caller will resume. Our benchmark driver reads the result off a root task if it has exited, and resumes it otherwise.

Using our **task<type>** class, we turned the relevant Masstree functions into nestable coroutines. This process is fairly simple:

- We added a suspension point (*i.e.,* `co_await suspend_always;`) after every prefetch. This turns functions that prefetch into coroutines.
- For every function (except the driver) that calls a coroutine, we inserted keyword **co_await** between the function call and its result. This turns callers into coroutines.



(a) 1 Thread Varying Group Size (b) 28 Threads Varying Group Size (c) 56 Threads Varying Group Size (NUMA) (d) Varying Thread

Figure 11: Masstree Experiments

- We changed the return type of every function that we turned into a coroutine from type **T** to **task<T>**.
- For every function that we turned into a coroutine, we replaced keyword **return** in its definition with **co_return**.

We also modified our benchmark driver so that each thread maintains an array of tasks, resuming each in turn. Adding coroutines to Masstree and modifying our benchmark driver took us around three developer hours. We compiled the result using Clang.

Benchmarks Setup. To see what effect coroutines had on Masstree performance, we ran the YCSB-B benchmark (95% reads, 5% writes) on 8-byte keys, with 8-byte values, using YCSB's Zipf distribution, where the distribution of read and write keys is skewed. We used a database of 250 million records.

4.1.2 *Vary Group Size*

Every workload will have a sweet spot of in-flight operations, where there are enough coroutines to cover memory latency from the slowest relevant level of the memory hierarchy, but not so many that they spill out of CPU cache. Since we are using a Zipf distribution, Masstree's leaf nodes that hold popular keys will be in one of the CPU caches (which level of CPU cache depends on how popular they are), while unpopular leaf nodes will have to be loaded from DRAM. The Zipf distribution's skew means that the optimal group size will be > 10, the number of line-fill buffers, since we can execute several prefetches from L3 cache while waiting for one prefetch from DRAM to complete.

In Figure 11(a) we vary the group size from 1 to 50 coroutines per thread, on a single hardware thread. The coroutines approach is able to achieve 3x the performance of an unmodified Masstree, with a group size of 30 to 50 coroutines per thread.

In Figure 11(b), we vary the group size from 1 to 200 coroutines per thread, on a full, single CPU. (The CPU has 14 cores, and we use all 28 of its hardware threads.) With coroutines, Masstree performance improves by 1.7x to 1.8x, from 12 to 200 coroutines per thread. Figure 11(c) shows the effects of non-uniform memory access (NUMA), as we fully use two CPUs. In this case, we see a local maximum at 12 coroutines per thread, with 1.4x the performance of an unmodified Masstree. Using 12 coroutines per thread hides memory latency within a single NUMA node. But we also see performance increasing to 1.6x to 1.7x beyond 150 coroutines per thread.

NUMA extends the trade-off one makes when varying the number of coroutines per thread. We used Intel VTune to explore this trade-off further; see Figure 12. Going from an unmodified Masstree to one that uses 30 coroutines per thread reduces the CPU time per operation by 27%, most of which can be attributed to reduced time spent accessing the CPU cache and DRAM. Going from 30 coroutines to 200 coroutines reduces the CPU time per operation by an additional 19%. As the chart shows, L2 and L3 stalls increase as DRAM stalls decrease. Because we have more coroutines, state is forced out of L1 cache into L2 and L3—but we are able to hide more DRAM latency, resulting in a net win. (Had Masstree prefetched using `_MM_HINT_NTA`, rather than `_MM_HINT_T0`, state forced out of L1 cache would have been evicted to DRAM, and we would not have seen a performance gain at 200 coroutines. The optimal hint, in this case, would have been `_MM_HINT_T1`, since our group size is large enough that the data we prefetched won't fit in L1 cache anyway, but we intentionally made no modifications to Masstree beyond adding coroutines.)

4.1.3 *Scalability with Number of Threads*

Figure 11(d) shows performance on a single CPU as the number of hardware threads ranges from 2 to 28. The threads are pinned so that every 2 threads fill a CPU core. Adding coroutines to Masstree yields performance between 1.7x (at 26 threads) and 2.1x (at 4 threads) that of an unmodified Masstree. The performance gain is still significant at 28 hardware threads, fully using all 14 cores of our test CPU. The stair-step "Coro" curve shows the interaction between coroutines and hyper-threading, which allows the CPU to schedule two hardware threads using the same CPU core. Hyper-threading gives some of the benefits of coroutines, and vice versa, but combining both yields the best overall performance.

4.2 Bw-tree

The Bw-tree is a latch-free B+-tree originally proposed by Microsoft Research [19], that has since been deployed in several Microsoft products and adopted by open source projects [2]. In this section, we evaluate the different interleaving approaches, using the Bw-tree as a representative of a class of highly-efficient, hardware-conscious B+trees. We tried adding coroutines to the Bw-tree in two different ways.

Our first approach was similar to AMAC, in that we moved all **get(key)** logic into a single function. Since the Bw-tree spends a significant amount of time stalled waiting for deltas to be brought into CPU cache, we modified our single **get()** function to suspend after prefetching the next delta.

Our second approach was to modify the Bw-tree the same way we modified Masstree, which we believe would perform significantly better than the first. However, we ran into a compiler bug in MSVC that prevents us from evaluating it; and, due to various dependencies, we were not able to compile the Bw-tree with Clang.

Unlike with Masstree, we also spent significant effort to add AMAC and GP to the Bw-tree. As an example, the resulting AMAC code has 10 stages, and does not look anything like the original, non-interleaved code we started with. Our first coroutines approach is a lot more readable, mostly resembling the non-interleaved code. This, once again, demonstrates the developer productivity benefits of coroutines, when applying interleaving techniques to state-of-the-art data structures.

[2]Available for download at: `https://github.com/cmu-db/peloton/tree/master/src/index`

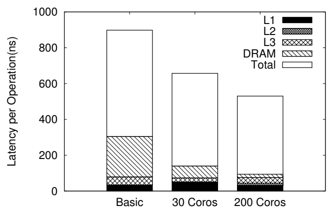
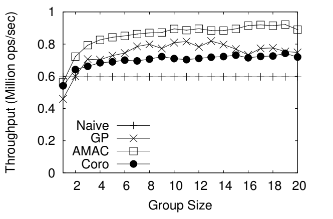
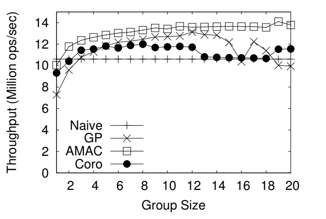


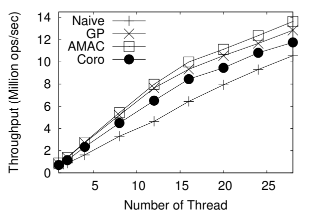
Figure 12: Masstree Microarchitectural Analysis



(a) 1 Thread Varying Group Size



(b) 28 Thread Varying Group Size



(c) 12 Group Size Varying Thread

Figure 13: Bw-Tree Experiments

We next present our results for Bw-tree using MSVC on Windows, using our first coroutines approach. We prefill the Bw-tree with 25 million records, each of which has an 8-byte key and an 8-byte value. We then run a random read write workload for 30 seconds, with 95% reads, 5% updates, using a uniform distribution, and measure the throughput. Each data point reported is an average of 3 runs.

4.2.1 Varying Group Size

In the first experiment, we want to study the impact of varying group size on throughput, first using a single thread and later expanding it to multi-threaded tests. For the figures in this section, we present the different group sizes on the x-axis, and the y-axis shows the throughput in million operations per second.

Figure 13(a) presents the single-threaded results. For the single group case, as expected, naïve is 1.3x, 1.06x, and 1.1x faster than GP, AMAC, and Coro, respectively. The reason is that interleaving only adds additional overhead without giving any benefits with a group size of 1. For group sizes >2, naïve is up-to 1.4x, 1.5x, and 1.3x slower than GP, AMAC, and Coro, respectively. Overall, AMAC is the fastest and is 1.43x, 1.15x, and 1.22x faster on average than naïve, GP, and Coro. As we have seen from both the hash probe and binary search experiments, we believe that the Coro approach, if optimized properly by the compiler, can achieve at least as much performance as the AMAC approach, without the added code complexity.

Figure 13(b) presents the results with 28 threads, and varying group sizes. In this case as well, AMAC beats all the other approaches and is 1.24x, 1.15x, and 1.17x faster on average than naïve, GP, and Coro, respectively. But as noted in the single-threaded case, given that AMAC is able to improve the throughput of Bw-tree by up-to 50% than naïve, we believe that the Coro approach can match that performance once the MSVC compiler issues are resolved.

4.2.2 Scalability with Number of Threads

Figure 13(c) presents the result of an experiment with varying number of threads from 1 to 28. We fix the group size per thread to 12 for all approaches, based on the results from the previous section. We see that GP, AMAC, and Coro scale better than naïve. And once again AMAC beats all the other approaches, by achieving an average of 1.5x, 1.1x, and 1.2x speedup over naïve, GP, and Coro, respectively.

In summary, for a state-of-the-art Bw-tree implementation, overall, AMAC performs the best, beating naïve by up-to 1.7x. This shows the potential of interleaving approaches to improve throughput for an already highly-optimized, hardware-conscious B+-tree. Notably, the Coro approach lags behind due to compiler issues. However, given the high performance of coroutines generated by the Clang compiler, as seen in the Masstree experiments, we expect to see similar gains for Bw-Tree.

5. DISCUSSION

The effectiveness of software prefetching relies on being able to do other work between the time when memory is prefetched and when it is read. In some cases (for example, when traversing an in-memory data structure—such as during a binary search or a hash join) data dependencies within an operation mean that there is no other work to do. Because subsequent CPU instructions, within the same traversal operation, need to read the memory that was prefetched, the CPU ends up stalling on the memory read after executing only a handful of instructions. In these cases, software prefetching works, and incrementally improves performance, but ultimately there is not enough distance between the *prefetch* and the *fetch* for it to have a large impact.

All of the techniques examined in this paper address this fundamental problem by performing several traversal operations simultaneously or by "interleaving". By doing so, the CPU can issue the next prefetch, for the next traversal operation, while waiting for the current prefetch to complete. One question is whether the requirement to "batch" multiple operations to exploit these techniques is realistic. Generally, in multi-tenant cloud environments, where many modern (database) systems operate, it is possible to batch requests from multiple users and exploit interleaving to improve overall system performance. More specifically, in [22], the authors present index joins for executing IN-predicate queries in SAP HANA as another example, where the values in the IN-predicate list are synonymous with a multi-get operation against the index. Batching is also common in other parts of a database engine. For example, some database systems already have a batch (instead of a tuple-at-a-time) interface between the query processing engine and the storage engine, which can be exploited for interleaving.

Another, inexpensive, way to hide memory latency is to make use of hardware threads provided by the CPU. In this paper, we used Intel CPUs, which offer only two hardware threads per CPU core. However, our experiments show that performance improves significantly, beyond two hardware threads, when we use coroutines. In other words, two hardware threads are not enough to hide memory latency for the workloads we tested. Using coroutines requires slightly more programmer effort than simply enabling (hardware or software) threads, but—as this paper and previous work [22] show—much less effort than a hand-rolled solution (GP or AMAC). In summary, coroutines offer a performance improvement, on widely-used CPUs, beyond two hardware threads with a fraction of the development effort.

This paper and previous work [18,22] show that using interleaving approaches significantly improves performance. Also, by using even a relatively small group size, one can achieve most of the performance benefit from interleaving approaches. We further note that the improvement can be overstated, on x64 CPUs, by disabling hyper-threading. We have shown in this paper that coroutines, in particular, improve performance significantly even when taking advantage of hyper-threading.

Table 3: Maximum Improvement over Naïve

	Compiler	Technique		
		GP	AMAC	Coro
Hash Probe	Clang	7.5	8.4	8.2
	MSVC	7.8	8.5	7.3
Binary Search	Clang	2.1	2.3	2.3
	MSVC	3.1	3.5	2.6
MassTree	Clang	-	-	3.2
Bw-tree	MSVC	1.7	1.7	1.4

Previous work [22] analyzes the impact of cache misses in detail, including TLB misses; we refer the interested reader to that paper for a detailed microarchitectural analysis of software prefetching using coroutines, and to [15] for an excellent background on prefetching in general. Note that [22] ran experiments on Intel's Haswell architecture, while we ran experiments on the successor architecture, Broadwell—and Kaby Lake CPUs are now available. There may be some differences in performance depending on the architecture used, which is to be expected.

We also note that previous work [22] has focused on MSVC's implementation of coroutines, but now a newer, more mature, implementation is available in Clang. In this paper, we compared MSVC coroutines to Clang coroutines and found that the latter currently offers much better performance (more below). It would be useful to repeat these experiments once MSVC improves optimizations specific for coroutines, which is a work in progress. As our results show, particularly for the naïve implementations, performance depends on the choice of compiler. And although we did not report them in this paper, we also ran all of our experiments with Clang on Linux, where performance was similar to Clang on Windows, but Coro was roughly 10% faster in that case.

In Section 3.1 and 3.2 we showed the benefits of the different interleaving approaches for simple data structures. Further, in Section 4.1, we showed that adding coroutines to Masstree—which is a relatively complex data structure–improved performance significantly, both on a single CPU and across NUMA. Adding coroutines to Masstree was straightforward and did not negatively affect the Masstree code base. To do this, we implemented a simple **task<T>** class, modeled on the similar class in C#. Using the **task<T>** class allowed us to suspend and resume chains of nested function calls, rather than just a single function. To the best of our knowledge, we are the first to demonstrate the use of coroutines for chains of nested function calls, which are common in practice.

Another practical consideration is the interaction between the coroutines-based code and existing components, such as a scheduler, of a (database) system. Our benchmark driver uses a simple round-robin scheduler. As a practical example, previous work [22] has already shown how to incorporate coroutines in SAP HANA to speedup IN-predicate queries. Furthermore, that work shows how simple "sequential" and "interleaved" schedulers can be implemented to run code sequentially or with coroutines (interleaved). The main idea is that when the expected benefit of interleaving is small, e.g., when there is not enough work to overlap, a sequential execution performs better. A key advantage of the coroutines-based approach is that by simply passing in an additional flag (true or false) to the index lookup operation, the same code can be made to run sequentially or interleaved, thus minimizing the impact on the rest of the system. This would not have been possible for AMAC or GP, since they require an almost complete rewrite of the code.

Last, Table 3 shows the *maximum* performance gain of a particular approach was able to achieve for a given data structure and compiler combinations in comparison to naive. We present the most important highlights below:

High Efficiency, High Developer Productivity: In terms of performance, the coroutines-based approach matches the performance of the AMAC-based approach when using Clang. Meanwhile, Coro beats naïve by up-to 3.2x in our Masstree experiment, with a trivial code change. This proves that for a wide variety of scenarios, we can reap all the benefits of "interleaving" and prefetching, without sacrificing developer productivity.

Performance Gains vs. Data Structure: Software prefetching significantly improves performance in all the scenarios we considered. However, the performance gain varies by data structure. For hash probe, binary search, Masstree, and Bw-tree we see a speedup of up-to 8.4x, 3.5x, 3.2x, and 1.7x, respectively, over naïve.

Compiler Support for Coroutines: As shown, coroutines support in the two popular compilers, namely Clang and MSVC, is not equally mature. Our results show that, in terms of its relative performance to other interleaving approaches, Clang-compiled Coro performs better than the MSVC. We draw two conclusions: (1) Clang's Coro performance proves that getting good performance (over hand-coded, hand-tuned AMAC code) is possible with compiler support, and (2) we are in early days for coroutines in C++. As the coroutines proposal for C++20 moves forward, making coroutines more main-stream, we expect the quality of coroutine code generation and the availability of coroutine libraries to improve. Therefore, we expect that Coro will be able to match or exceed the performance of handcrafted interleaving solutions, in the future.

6. CONCLUSION

In this work, we started with the goal of implementing and evaluating coroutines-based data structures to hide memory stall latency. We presented an in-depth evaluation of coroutines-based approaches for hash probe, binary search, Masstree, and Bw-tree. We compared no-prefetch (naïve) with existing state-of-the-art software prefetching approaches namely GP and AMAC. We show that all of these approaches perform significantly better than the naïve approach. In some cases (e.g., hash probe), these approaches are better by a *factor of 8*. We also confirm that with coroutines, in most cases, we can match or beat the performance of hand-written, hand-tuned AMAC (or GP) code at a tiny fraction of the development cost. And our implementation and evaluation verify the applicability of coroutines- based approaches across simple and complex data structures. Further, to the best of our knowledge, we are the first to demonstrate multi-function coroutines, and to evaluate the effectiveness of coroutines to hide memory latency when hyperthreading is turned on and across NUMA nodes. Finally, we summarized our findings, which provide a guideline for system builders to exploit coroutines in designing highly efficient systems.

7. REFERENCES

[1] The Clang project. `https://clang.llvm.org/`.

[2] Coroutines ISO. `https://www.iso.org/standard/73008.html`.

[3] CppCoro. `https://github.com/lewissbaker/cppcoro`.

[4] Fibers. `https://msdn.microsoft.com/en-us/library/ms682661.aspx`.

[5] Intel 64 and IA-32 architectures optimization reference manual. `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`.

[6] Microsoft Visual Studio. `https://www.visualstudio.com/`.

[7] Programming languages—C++ extensions for coroutines. proposed draft technical specification ISO/IEC DTS 22277 (e). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf.

[8] Redis. https://redis.io/.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *PVLDB*, pages 266–277, 1999.

[10] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. 11(5):553–565, 2018.

[11] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *CACM*, 60(4):48–54, 2017.

[12] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *TODS*, 32(3):17, 2007.

[13] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[14] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[15] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[16] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. *NSDI*, 13:371–384, 2013.

[17] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.

[18] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[20] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database achitecture for the new bottleneck: Memory access. *VLDBJ*, 2000.

[21] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, 2012.

[22] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *PVLDB*, 11(2):230–242, 2018.

[23] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, 2000.

[24] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for DRAM-based storage. In *FAST*, 2014.

[25] V. Sikka, F. Färber, A. Goel, and W. Lehner. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *PVLDB*, 6(11):1184–1185, 2013.

[26] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 2013.