LRU-C: Parallelizing Database I/Os for Flash SSDs

Bohyun Lee
Sungkyunkwan University
lia323@skku.edu

Mijin An*
SAP Labs Korea
mijin.an@sap.com

Sang-Won Lee
Sungkyunkwan University
swlee@skku.edu

ABSTRACT

The conventional database buffer managers have two inherent sources of I/O serialization: read stall and mutex conflict. The serialized I/O makes storage and CPU under-utilized, limiting transaction throughput and latency. Such harm stands out on flash SSDs with asymmetric read-write speed and abundant I/O parallelism. To make database I/Os parallel and thus leverage the parallelism in flash SSDs, we propose a novel approach to database buffering, the *LRU-C* method. It introduces the LRU-C pointer that points to the *least-recently-used-clean* page in the LRU list. Upon a page miss, LRU-C selects the current LRU-clean page as a victim and adjusts the pointer to the next LRU-clean one in the LRU list. This way, LRU-C can avoid the I/O serialization of read stalls. The LRU-C pointer enables two further optimizations for higher I/O throughput: *dynamic-batch-write* and *parallel LRU-list manipulation*. The former allows the background flusher to write more dirty pages at a time, while the latter mitigates mutex-induced I/O serializations. Experiment results from running OLTP workloads using MySQL-based LRU-C prototype on flash SSDs show that it improves transaction throughput compared to the Vanilla MySQL and the state-of-the-art WAR solution by 3x and 1.52x, respectively, and also cuts the tail latency drastically. Though LRU-C might compromise the hit ratio slightly, its increased I/O throughput far offsets the reduced hit ratio.

PVLDB Reference Format:
Bo-Hyun Lee, Mijin An and Sang-Won Lee. LRU-C: Parallelizing Database I/Os for Flash SSDs. PVLDB, 16(9): 2364 - 2376, 2023.
doi:10.14778/3598581.3598605

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LeeBohyun/LRU-C.

1 INTRODUCTION

The database buffer manager interacts with the underlying storage with read and write operations. In conventional relational database systems, numerous concurrent processes are involved in making such I/O requests. For instance, foreground processes issue read operations to fetch missing pages, and background flushers make write requests to flush dirty pages for durability and checkpoint. Unfortunately, conventional database buffer managers have two inherent sources of I/O serialization when concurrent processes issue their I/Os: *read stall* and *mutex conflict*. The I/O serialization barely harms on hard disk with slow and symmetric read-write speed and low parallelism. On flash SSDs with asymmetric read-write speed and abundant I/O parallelism, however, the peril of serialized I/Os stands out, making storage and CPU under-utilized and thus limiting transaction throughput and latency [1, 2]. For this reason, modern database systems still experience the problem of I/O serializations when running on flash SSDs.

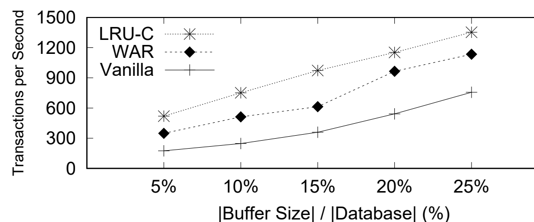


Figure 1: TPC-C Throughput: Vanilla, WAR [2], and LRU-C

It is well known that upon buffer misses foreground processes suffer from the *read stall* problem on flash storage with read-write asymmetry [2]. That is, the read operation by a foreground process is stalled until the slow write operation completes to secure a clean frame [40]. Though those two I/O operations of read and write can proceed in parallel, they have to be serialized just because they happen to share the same buffer frame. The read stall problem is a type of *RW-serialization*.

Another source of I/O serializations is the *mutex conflict* at the buffer manager layer. The buffer manager taking the LRU (Least Recently Used) replacement policy or its variant employs a single mutex to protect the consistency of the LRU list from the concurrent accesses by multiple threads [12]. Each foreground process and background flusher must obtain the LRU mutex first and hold the mutex until the I/O operation ends. This protocol for acquiring and releasing the LRU mutex will serialize the read I/Os among foreground processes. As a foreground process scans the LRU tail while holding the mutex, other processes have to wait for the mutex. We call the mutex-induced ordering of read requests as *RR-serialization* in the paper. In addition, as discussed later, the LRU mutex also forces to serialize read and writes between foreground processes and background writers (i.e., another type of *RW-serialization*).

I/O serialization elicits adverse effects on system utilization and transaction performance on flash storage. First, foreground processes waste CPU time while simply waiting for the preceding operation to complete. Second, I/O serialization at the buffer layer blocks the opportunity for parallel processing in storage, even when their read or write operations are likely to target the different flash channels/ways and thus proceed in parallel. Lastly, I/O serialization worsens the transaction latency. Because synchronous reads are in the critical path of an in-progress transaction [13], the time taken

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598605

* Work done while in Sungkyunkwan University.

to finish the preceding I/O(s) is added to the critical path of the I/O-serialized transaction, which prolongs its latency.

To make database I/Os parallel and thus to better leverage the parallelism in flash SSDs, we propose a novel approach to database buffering, called the *LRU-C* method. It introduces the LRU-C pointer, which is used to point to the *least-recently-used-clean* page in the LRU list. With the list implementation of the LRU policy, the LRU-clean page will refer to the first clean page from the LRU tail. Upon page miss, LRU-C selects the LRU-clean page as the victim and adjusts the LRU-C pointer to the next LRU-clean page. This way, a page miss can be instantly served using the LRU-C page. As a result, LRU-C avoids the read stall and minimizes the scanning overhead in the LRU tail. All the dirty pages between the pointer and the LRU tail are periodically flushed as victims by the background flusher.

The idea of LRU-C is not about a new victim selection policy but a new I/O architecture between buffer manager and flash storage. The main contributions of this paper are summarized as follows:

- We identify two sources of I/O serializations in the conventional database buffer manager, *read stall* and *mutex conflict*, which severely limit transaction throughput and latency, especially on flash storage with asymmetric read-write speed and parallelism.
- As a principled approach to mitigate these I/O serializations, we propose LRU-C. It introduces the LRU-C pointer to quickly select the *least-recently-used-clean* page from the LRU list for replacement. Another critical role of the LRU-C pointer is that it logically divides the LRU list into two distinct regions, *mixed* and *dirty*. The idea of the LRU-C pointer is simple, straightforward, yet novel and effective.
- The introduction of the LRU-C pointer enables two further optimizations for higher I/O throughput: *dynamic-batch-flush* and *parallel LRU-list manipulation*. With dynamic-batch-flush, all dirty pages accumulated so far between the LRU-C pointer and the LRU tail are flushed altogether, which makes the write batch size dynamic and larger than before. Moreover, as the LRU-C pointer divides the LRU list into two regions, we introduce another mutex, `LRU-D` to protect the dirty region. The original LRU mutex is now used to synchronize the mixed region. This way, LRU-C allows the foreground and the background threads to manipulate LRU concurrently and thus issue their read and write requests in parallel to the storage.
- We prototype LRU-C by modifying the MySQL codebase minimally. As shown in Figure 1 obtained from running the TPC-C benchmark on flash SSD, LRU-C can outperform the Vanilla MySQL (i.e., an I/O-tuned MySQL, as detailed in Section 5.2.1) and the state-of-the-art WAR solution [2] by 3x and 1.52x, respectively. This indicates that LRU-C can effectively resolve mutex collisions as well as read stalls, making database I/Os truly parallel.

2 BACKGROUND

In this section, we explain two characteristics of flash memory SSDs and describe the I/O architecture taken by the conventional database buffer managers in detail. Then, we discuss three types of I/O serializations inherent to the database I/O architecture.

Table 1: I/O Speed Asymmetry in Storage Devices

Storage Device	Random IOPS (16KB) Read	Write	Asym. Ratio ($Read/Write$)
SSD-A†	190,622	33,866	5.6
SSD-B¶	169,681	33,811	5.0
SSD-C⋄	39,860	3,830	10.4
SSD-D*	80,433	12,536	6.4
HDD♯	280	216	1.3

† Intel P4101 NVMe SSD 1TB, ¶ Samsung 970Pro NVMe SSD 512GB, ⋄ Micron Crucial MX500 250GB, * WD Blue SN570 500GB, ♯Western Digital WD10EZEX 1TB

2.1 I/O Asymmetry and Parallelism in SSDs

Asymmetric Read-Write Speed The read and write speeds of NAND flash memory are asymmetric because it takes longer to write a page than to read a page from flash memory chips. In addition, the costly but inevitable garbage collection operations further widen the gap between the two [45]. To assess the read/write asymmetry in flash memory SSDs, we measure the random IOPS (I/Os per Second) of four commercial SSDs and one hard disk by running the FIO benchmark [4] with the queue depth of 32 for 30 minutes against each device with half-full of data. Table 1 presents the result. The asymmetric ratio is given in the last column of the table, calculated by dividing the read IOPS by the write IOPS. The read IOPS in SSDs is typically at least five times larger than the write IOPS. In contrast, the speed of read and write is almost the same in HDD. To summarize, asymmetry ratios can vary for individual devices, but it is a unique characteristic of every flash SSDs.

Parallelism Flash storage processes I/O requests by using internal parallelism at various levels [7, 25]. NAND flash package consists of multiple chips, and a single chip is formed by multiple planes. The flash controller communicates with the flash packages through multiple channels. Channels are shared by multiple packages and can be accessed at the same time [26]. This way, blocks can be accessed simultaneously across different chips. As a result, flash storage can perform multiple I/O requests once in parallel. As such, it is always desirable for the host system to allow different threads to make their I/O requests in parallel, thus leveraging the abundant parallelism in flash storage.

2.2 Database I/O Architecture

One prominent role of the buffer manager is to quickly age out the rarely used pages from the buffer cache while keeping frequently accessed pages cached. To leverage the locality in data accesses and thus maximize the efficiency of limited cache size, the buffer cache is usually implemented as a linked list of buffer frames and managed according to its buffer replacement policy (e.g., LRU) [38].

When a foreground thread encounters a page miss, it first has to obtain a free buffer frame to which the missing page will be read. Below we detail the steps the foreground process follows at this point as the background for our work. Throughout this paper, LRU is assumed as the buffer replacement algorithm for the simplicity of discussion. However, the problem of I/O serialization is equally

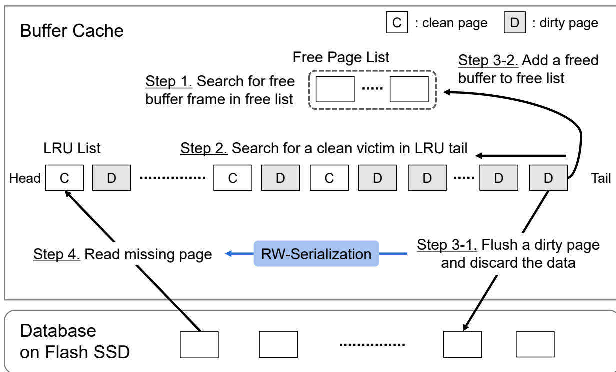


Figure 2: I/O Architecture in Database Buffer [2]

applicable to other buffer replacement policies without loss of generality. Let us explain the overall process of buffer replacement using Figure 2, focusing on the process of evicting dirty pages.

Page Read by Foreground Processes If a user thread requests a free buffer frame to read a page from the disk, the buffer manager goes through three stages to secure a free buffer frame. First, it searches for a free buffer frame in the free list (Step 1). If there is an available free buffer frame, it is used for read immediately. Otherwise, if no free buffer is available from the free list, a victim buffer frame has to be secured. To do so, the foreground thread first scans the LRU list from the tail until a predefined scan depth to find a clean buffer frame (Step 2). If a clean frame is not found in the tail, a dirty page at the LRU tail is chosen for eviction, and the foreground thread waits until the page is written to the disk (Step 3-1). Once a free frame is added to the free list (Step 3-2), the read request can finally be issued (Step 4). In this architecture, evicting a dirty page costs disk write, so it is better to choose a clean page within a predefined scan depth whenever possible when considering transaction response time.

Page Write by Background Flushers Writes to the database are asynchronous in nature. This is because database durability is guaranteed by forcing redo logs at commit and many database engines take a no-force policy for buffer management [19]. Therefore, many database engines are equipped with background flushers to take advantage of asynchronous writes [5, 9, 22, 42]. They use Linux AIO to pre-flush dirty pages from the LRU tail. By pre-flushing dirty pages, they aim to free frames in advance so that each foreground process can immediately get a free frame from the free list upon a buffer miss. As a result, foreground threads consume free pages in the free list to execute read requests, whereas background flushers periodically produce free frames.

LRU Mutex and Contention Buffer frames in the LRU list are frequently repositioned. For example, the foreground process moves the hit page to its head on a page hit. In addition, the background flushers move dirty pages at the LRU tail to the free list after flushing them. Therefore, multiple threads share and manipulate the LRU list concurrently, including foreground processes and background flushers. To protect the consistency of the LRU list, multi-threaded DBMSs employ a global mutex, LRU mutex [21]. This mutex is a synchronization mechanism to enforce that only one thread at a given time can have access to the shared resource [12]. Therefore, every thread must obtain the LRU mutex before accessing the LRU list. Meanwhile, when a thread attempts to obtain an already locked mutex, mutex contention occurs, and the thread must wait for the other thread to release the mutex. This mutex contention can lead to the degradation of transaction throughput.

2.3 I/O Serializations

Based on the description above, this section discusses two types of I/O serialization: RW (Read-Write) and RR (Read-Read) serializations and two reasons behind them. Recall that serialized I/O requests cannot fully utilize the parallelism of flash storage.

RW-Serialization due to Read Stalls In Section 2.2 and Figure 2, it is explained that if a free buffer frame cannot be secured during Step 1 or Step 2, the foreground process is required to obtain a free buffer frame by the RAW (Read After Write) protocol [2] (i.e., Step 3-1). This protocol involves reading a missing page after writing a dirty page at the LRU tail, and therefore results in a read stall [2], where the read operation is stalled until the write operation completes. This paper defines a read stall as one of the *RW-serializations*. When a read stall occurs, the foreground process issues a page write, and then a page read in a serialized order.

To quantitatively verify the cost of read stalls, we modified the MySQL source code to measure the time duration that each foreground process has to wait upon a read stall. Then, while running the TPC-C benchmark with the same configuration as in Table 2 for 30 minutes, we collected the wait times of all read stalls of more than five million and then computed their average wait time. The average wait time of all read stalls was 4.86 ms, and the worst wait time was more than 500 ms. Note that the read latency in flash devices is usually several hundred microseconds. Provided that the time taken to flush dirty pages upon read stalls will be in the critical path of the transaction execution, the cost of read stalls will exacerbate transaction latency and throughput [2].

Mutex-Induced RW- and RR-Serializations Splitting a contended mutex is a standard practice for improving the parallel performance of multi-threaded programs. Therefore, there have been many efforts to improve the scalability of the DBMS buffer cache. For instance, MySQL/InnoDB [9] has recently divided the global `buffer_pool_mutex` mutex protecting the whole buffer pool into three mutexes, `lru_list_mutex`, `flush_list_mutex`, and `free_list_mutex`. Despite the mutex separation, the LRU mutex still remains as the main source of mutex contention, hindering the I/O parallelism among processes. In a separate experiment, we measured the degree of LRU mutex contention and confirmed that a thread waits for about 1.3% of benchmark runtime to acquire the LRU mutex. Considering the speed of DRAM operations, 1.3% is never a negligible figure. Furthermore, among all other mutex types, mutex contention occurs the most in the LRU list.

Such mutex contention frequently occurs when the foreground and background threads compete for the same LRU mutex, especially in OLTP workloads with intensive writes. As depicted in Figure 3, although the background flusher attempts to write dirty pages in the LRU tail asynchronously in time, the LRU list is often

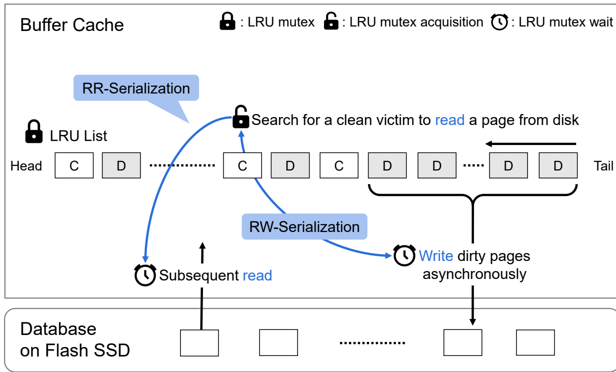


Figure 3: Mutex-induced I/O Serializations

occupied by foreground threads handling user transactions. Consequently, the background flusher has to, although it works in the background, wait for a foreground process to release the mutex. That is, *RW-serialization* occurs when read and write requests issued by the foreground and background processes are serialized due to the shared mutex. Additionally, *RR-serialization* occurs when foreground processes contend for acquiring the LRU mutex, as illustrated in Figure 3. As a foreground process searches for a clean victim page to replace with a page it is about to read, other foreground processes wait for the LRU mutex to be released to serve subsequent page read operations. Likewise, numerous processes compete for the LRU mutex acquisition to access pages, especially in the LRU tail. Such serialization worsens as the number of simultaneous users increases.

3 MOTIVATION: I/O SERIALIZATIONS ON SSDS

Although the existing database systems have such sources of I/O serializations, they will rarely experience I/O serializations on top of hard disks due to the slow and symmetric read-write speed and the low parallelism.

First, foreground processes do not encounter the read stall situation. Because of the symmetric read-write speed in hard disks, the background flusher can produce the free buffers with a similar pace to which the foreground consumes the free page [2]. Thus, foreground processes can obtain a frame to read the missing page from the free page list or the LRU tail scan. In addition, the hard disk has no or low internal parallelism [6]. Due to its slow mechanical read/write speed, the penalty of mutex contention barely impacts transaction throughput. Hence, the performance gain from parallelizing mutex-induced I/O serialization is limited. As will be investigated in Section 5.2.7, the performance gain of LRU-C over the Vanilla is very marginal when hard disks are used as a storage device (refer to Figure 13). This result implies that threads hardly experience serializations on the hard disk. We guess that for this reason, the potential risk of I/O serializations remains unaddressed with the existing database systems.

In contrast, on flash SSDs with fast but asymmetric read-write speed and abundant I/O parallelism [6], the potential risks of I/O serializations stand out, making storage and CPU under-utilized and thus limiting transaction throughput and latency. First, foreground

Table 2: TPC-C Evaluation on MySQL/InnoDB

Buffer Size / DB Size (%)	5	10	15	20	25
Read Stall (%)	29	33	31	17	5
TPS	175	247	359	542	756
95% Latency (ms)	450	332	257	194	152
CPU Util. (%)	26	25	28	35	41

processes frequently encounter read stalls on flash SSDs [1, 2, 40] due to the asymmetric read-write speed. This is because the read operation is processed much faster than the write operation, leading foreground processes to quickly consume clean and free pages in the event of page misses, while the background flusher lags behind in producing free buffer frames. The read stall problem is not MySQL-specific but common across most relational DBMSs, including Oracle, PostgreSQL, XtraDB, and Zero DBMS [1, 2, 40]. Although the database writes are basically asynchronous with the help of background flusher, database systems will inevitably experience read stalls on flash storage.

Second, the mutex-induced RR- and RW-serializations severely under-utilize the abundant parallelism inside flash SSDs. For instance, while the background flusher writes dirty pages at the LRU tail, SSD can handle the random reads from foreground processes in parallel. However, with the global `LRU-mutex`, the foreground process has to wait until the mutex is released [40]. Thus, the read and write are serialized due to the mutex, which decreases the SSD utilization accordingly. A similar argument can be made for foreground processes that compete for the mutex before scanning the LRU tail in search of a clean victim page.

To investigate the impact of excessive serialized I/O requests on various buffer pool sizes, we measure the ratio of read stall along with three other performance metrics shown in Table 2 while running the TPC-C benchmark with 1,000 warehouses and 64 concurrent clients using MySQL/InnoDB on `SSD-A`. According to the second row of Table 2, 15-33% of page reads experience read stall when the buffer ratio (i.e., the buffer cache size divided by the database size) is below 20%. When the buffer ratio is larger than 20%, on the other hand, the relative number of read stalls reduces inversely due to an increased hit ratio [2]. As the number of read stalls decreases, TPS (Transactions Per Second) and tail latency improve, as indicated in the third and fourth row of Table 2. The last row of Table 2 shows how CPU utilization changes as buffer pool size increases. Contrasting with the read stall ratio in the second row, it is clear that the CPU usage is inversely proportional to the read stall ratio. This is because the CPU idly waits for the write operation to end upon read stall. Additionally, the CPU's time is wasted when foreground processes repetitively scan dirty pages from the LRU tail, exacerbating RR-serialization due to increased LRU mutex wait time. In summary, I/O serializations degrade performance and resource utilization.

4 DESIGN OF LRU-C

This section proposes a novel buffering architecture, LRU-C. Its goal is to remove three I/O serializations among concurrent database threads, which are discussed in Section 2.3, and thus to allow those

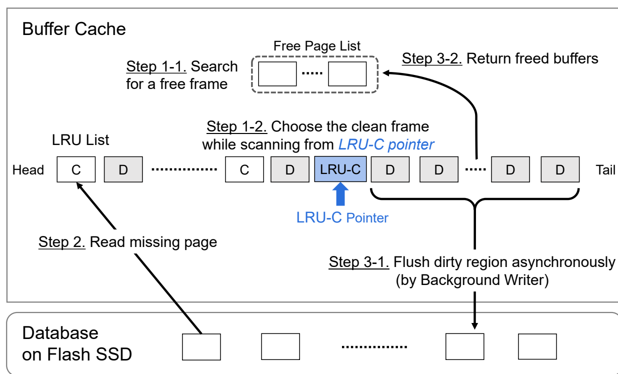


Figure 4: LRU-C: Design Overview

Algorithm 1 The LRU-C Buffering Algorithm

1: LRU-C: the oldest clean page in LRU
2: LRU-C Pointer: the pointer to LRU-C
3: **Output:** P_v : a victim page
4: **function** *select_victim*
5: Get LRU mutex
6: **if** LRU-C is dirty **then**
7: Update LRU-C pointer to point to next LRU-C
8: **end if**
9: P_v = LRU-C
10: Update LRU-C pointer to point to next LRU-C
11: Release LRU mutex
12: **return** P_v
13: **end function**

threads to issue their I/O requests in parallel. We explain its key idea, basic architecture, and optimizations. We also discuss its drawback in terms of hit ratio and describe the prototype implementation.

4.1 Key Idea

LRU-C stands for *'Least-Recently-Used-Clean'*, and the whole system of LRU-C operates around the concept of the *LRU-C pointer*. The pointer holds the clean page, least recently used among all clean pages (i.e., not dirtified). In the case of the list-based LRU implementation, it points to the first clean page from the LRU tail.

The pointer provides two immediate benefits with regard to victim selection on page miss. First of all, the LRU-C pointer removes the read stall problem. Unlike the RAW protocol described in Section 2.2, the foreground reader process can immediately obtain clean victim even when numerous dirty pages are left behind the LRU-C pointer. Thus, it prevents foreground processes from encountering the *RW-serializations* due to the read stall. Second, the LRU-C pointer allows the foreground processes to find the clean page more quickly (that is, just traversing only a few frames from the pointer) during Step 2 in Figure 2. The shortened scanning process enables the foreground process to release the LRU mutex earlier with less CPU consumption. Accordingly, other foreground threads can obtain the mutex earlier due to mitigated mutex contention. Consequently, LRU-C minimizes the *RR-serialization* among foreground processes with long in-between intervals due to the overhead of scanning the long LRU tail.

Some readers might think that LRU-C is similar to the CF-LRU replacement policy [36] in a way that both schemes prefer clean pages as victims. However, we would like to stress that LRU-C is not just a victim selection but an architecture for parallelizing DB I/Os. See Section 6 for detailed comparison between them. To summarize, despite the simplicity of LRU-C design, the introduction of the LRU-C pointer enables disentangling various I/O serializations inherent in the existing buffer management schemes. Moreover, we also extend LRU-C with further optimizations to truly parallelize read and writes for flash storage, which is detailed in Section 4.3.

4.2 Architecture

Now, let us explain how foreground threads handle page misses and the background flusher writes dirty pages. Upon page miss, each foreground process follows the procedures explained in Figure 4 and Algorithm 1 to secure a free buffer frame for page read. First, if any free buffer frame is available in the free page list, it simply returns one of the free buffer frames (Step 1-1). However, if the free page list is empty, it must select a victim page to replace the requested page. LRU-C always chooses a clean page as a victim by directly using the LRU-C page pointed by the LRU-C pointer (Step 1-2). After using the victim, the LRU-C pointer needs to be adjusted to secure another clean page for the next replacement. To do so, we scan the LRU from the LRU-C pointer until it meets the next clean page. The LRU-C pointer is also readjusted if the LRU-C is dirty. As such, the mixed region shrinks while the dirty region grows as the pointer moves toward the LRU head.

Meanwhile, the background flusher is responsible for flushing the dirty pages in the dirty region after the LRU-C point. As illustrated in Step 3-1, the background flusher process periodically (e.g., once every second) flushes the dirty pages at the LRU tail using the asynchronous I/O primitives. The background write is assumed to flush the fixed number of pages (e.g, the value of the `LRU-scan-depth` parameter in MySQL which is 1,024 by default). In this respect, we call this type of flushing as *basic LRU-C*. Once all dirty victims are flushed, background flusher returns free buffers to the free page list (Step 3-2).

4.3 Two Optimizations

The introduction of the LRU-C pointer brings two further optimization opportunities for higher I/O throughput: *dynamic-batch-write* and *parallel LRU-list manipulation*.

Dynamic-Batch-Write In basic LRU-C, more dirty pages tend to be stacked in the LRU tail than before. This is because the foreground processes consume clean pages more quickly without encountering read stalls. However, as the background flusher only writes a static number of dirty pages in the dirty region, it underutilizes the parallelism in flash storage. Moreover, the background flusher cannot produce enough free buffer frames for the foreground read process, as the limited free frames are returned to the free list. Likewise, since the background flusher cannot keep up with the pace of foreground read processes, LRU-C pointer approaches the LRU head faster while dirty pages left behind the pointer remain

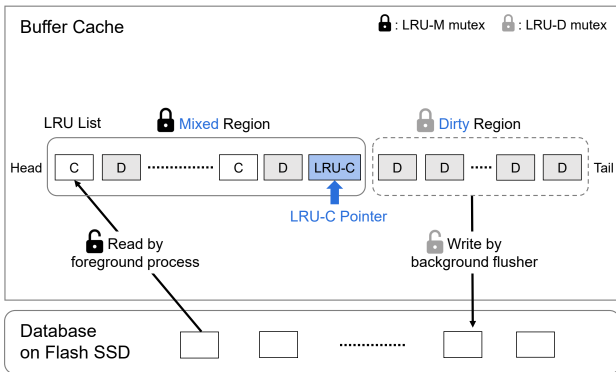


Figure 5: Parallel LRU List Manipulation in LRU-C

unused. In the worst scenario, the LRU-C pointer reaches the LRU head. In such cases, read stalls must occur.

Meanwhile, considering that the dirty pages in the dirty region have to be flushed sooner or later, there is no reason to flush only the fixed number of pages. Thus, whenever a background flusher is triggered, we decided to flush all dirty pages left behind the LRU-C pointer, instead of a fixed number of dirty pages. As a result, the number of flushed pages will *dynamically* vary depending on the location of the LRU-C pointer. That is, the background flusher takes the *dynamic-batch-write* approach to keep pace with the read speed. With the dynamic-batch-write optimization enabled, all dirty pages accumulated so far in the dirty region are flushed altogether, which makes the write pattern in LRU-C dynamic and larger.

This technique has two benefits. First, this will increase the write bandwidth and make the SSD utilization higher [15]. Second and more importantly, it will make the cold and dirty pages at the LRU tail at the earliest time free and available to foreground processes. As such, on page misses, more foreground processes can immediately obtain a free frame from the free list, instead of scanning the mixed region in search of a clean victim page. Recall that discarding such clean pages in the middle of the LRU list can reduce the hit ratio [36].

Parallel LRU-List Manipulation In the existing database buffers, one global mutex, called *LRU-list*, is usually used to protect the consistency of the LRU list from concurrent accesses of multiple threads. This mutex is a well-known point of contention for I/O-intensive OLTP workloads [21], mainly because background flusher and foreground threads compete for the mutex. In a separate experiment, we observed that even when the background flusher is triggered more aggressively, the transaction throughput does not improve due to the mutex-induced RW-serializations between the background flusher and foreground processes. Unfortunately, the mutex contention will also limit the impact of *dynamic-batch-write* optimization, as flushing more dirty pages will force foreground processes to wait longer for the global LRU mutex.

Figure 5 shows that the LRU-C pointer logically divides the LRU list into two regions: mixed and dirty regions. The dirty region refers to the portion of the LRU list from the LRU-C pointer to the LRU tail. The region consists of only dirty pages which are not recently accessed and thus are soon to be evicted. The mixed region refers to the rest of the LRU list, which consists of clean and dirty pages. The foreground read thread selects a clean victim from the mixed region, while the background flusher writes all the dirty pages in the dirty region. In other words, the foreground processes and the background flusher will access only mixed and dirty regions of the LRU list, respectively. Therefore, even when two different threads manipulate mixed and dirty regions concurrently, the LRU list still remains consistent since the foreground process and the background flusher do not invade the other's region. Nevertheless, they compete for the same mutex in order to start the operation.

Based on this observation, we decide to split the global LRU-list mutex into two mutexes, *LRU-M* and *LRU-D*, which cover two disjoint regions of the LRU list chain. The *LRU-D* mutex is used to protect the dirty region, while the *LRU-M* mutex is to protect the mixed region. With the help of two mutexes, two regions of the LRU list can be manipulated in parallel. Thus, we call this optimization technique as *parallel LRU-list manipulation*. The main objective of the optimization is to alleviate mutex contention between foreground processes and the background flusher. As will be illustrated in Section 5, it can considerably increase the transaction throughput. To sum up, LRU-C is, to our best knowledge, the first work which allows to split the LRU mutex for better I/O parallelism.

Let us explain how two mutexes work at their regions in the LRU list. The *LRU-M* mutex protects the mixed region ahead of the LRU-C pointer, while the new *LRU-D* mutex manages only the dirty region behind the pointer. In rare occasions, the foreground thread needs to acquire the *LRU-D* mutex when a page hit occurs against a dirty page in the dirty region. Except for such rare cases, read and write requests can be issued simultaneously by the foreground processes and the background flusher, as depicted in Figure 5. Thus, LRU-C can resolve the *RW-serialization* between them. Each thread can concurrently manipulate the mixed and dirty region using the *LRU-M* mutex and the *LRU-D* mutex, respectively. As a result, the foreground reader and the background flusher can issue their read and write I/Os independently in parallel. Also, note that the benefit of short-scanning in LRU-C will be more outstanding as foreground processes are no longer interfered with by the background flusher.

Despite the optimization of parallel LRU-list manipulation, LRU-C is not free from RW-serialization: it incurs another RW-serialization due to the checkpointing. Once the background flusher completes flushing dirty pages behind the LRU-C pointer, it then performs checkpointing afterward. To keep track of dirty pages to be checkpointed, most DBMSs, including MySQL, Oracle, and PostgreSQL, employ a dirty page list sorted by LSN (Log Sequence Number), and the background flusher writes dirty pages from the head of the dirty page list in small batches [10, 24, 39]. While dirty pages are mostly located in the dirty region of the LRU list, they can also be found in the mixed region. Therefore, to checkpoint those dirty pages in the mixed region, the background flusher in LRU-C also has to acquire the *LRU-M* mutex, and thus it has to compete for the mutex with the foreground processes. In this way, LRU-C also causes RW-serializations. However, note that the peril of checkpoint-induced I/O serialization is insignificant. In many cases, the background flusher will release the *LRU-M* mutex quickly because most dirty pages to be checkpointed have already been flushed while flushing the LRU tail and only a small number of dirty pages to be checkpointed remains in the mixed region.

4.4 Reduced Hit Ratio in LRU-C

On a page miss, LRU-C selects the LRU-C page instead of the dirty page at the tail. This could *reduce the hit ratio* since the LRU-C page is more likely to be accessed soon again than the tail page [36], according to the principle of temporal locality. However, the hit ratio gap between LRU-C and pure LRU is, as shown in Section 5, small in practice. Moreover, the loss in hit ratio will be far outweighed by the I/O performance benefit of LRU-C. Also, recall that the most least-recently-used dirty pages at tail will be flushed as victim every second by the background flusher.

4.5 Prototype Implementation

In order to demonstrate its effect, we prototype LRU-C by modifying the MySQL/InnoDB engine. The code modification are made only at the buffer manager module (`buf`), and less than 300 lines of new code are added. Three key changes made to the buffer module are as follows. First, we add the `LRU-C` pointer and the dirty region mutex, `LRU-D`, as new data structures to the `buf_pool` instance. Second, the victim selection algorithm is altered to search the LRU-clean page along the LRU list starting from the LRU-C pointer to the LRU head. Third, we make two modifications for the `page_cleaner_thread` to embody two additional optimizations of LRU-C. Prior to flushing dirty pages, it acquires the `LRU-D` mutex (for the optimization of parallel LRU-list manipulation). Then, it flushes all pages in the dirty region (for the optimization of dynamic-batch-write). When the page cleaner finishes its job, it releases the `LRU-D` mutex.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup and Workloads

All the experiments are conducted on a Linux system. We use a computing platform with 64GB of main memory and Intel Xeon Silver 4216 CPU with 32 cores. As the database storage, four commercial SSDs from Table 1 are used. We also use a Samsung 850 PRO 256GB SSD as the database log device in all experiments. All NVMe SSDs including `SSD-A`, `SSD-B`, and `SSD-D` are connected to the host via PCIe interface and `SSD-C` is connected via SATA interface. For MySQL server, we use the ext4 file system. The benchmark clients are run with database processes on the same computing platform. In all experiments, the database size is 100GB, the database page size is 16KB, and the number of concurrently running client threads is set to 64. For each experiment, we initialize the database storage device using the `dd` command and then run the benchmark for 30 minutes after a 5-minute warm-up time. Each performance result is the average result of three runs. We use two OLTP benchmarks, TPC-C and LinkBench.

- **TPC-C:** To benchmark TPC-C workload, we use tpcc-mysql [37] by Percona. The TPC-C variant enables performing database workload replay and industry-standard benchmark testing on MySQL. We use a 100GB database (i.e., 1,000 warehouses).
- **LinkBench:** LinkBench [3, 17] is an open-source database benchmark that simulates the social graph database workload on MySQL. The generated database consists of 100 million nodes, which is approximately about 100GB.

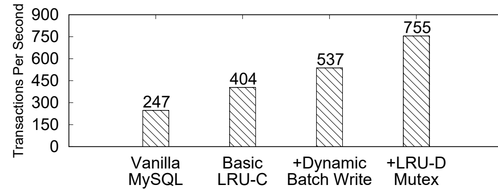


Figure 6: TPC-C Throughput: Vanilla vs. LRU-C Versions

Table 3: Read Stalls and Mutex Waits in LRU-C (TPC-C)

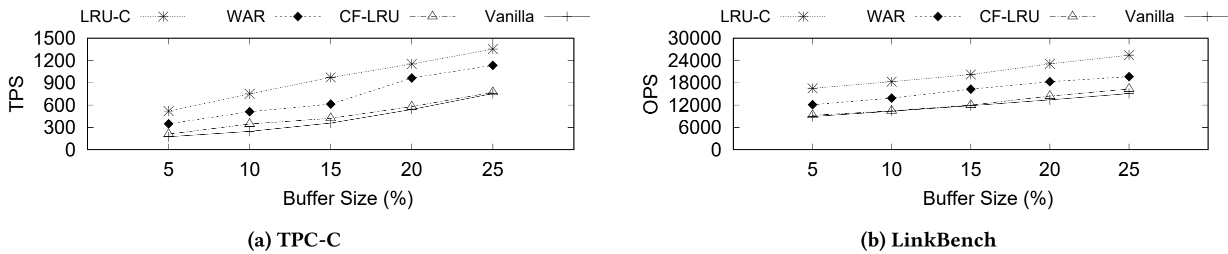
	Read Stall (%)	LRU Mutex Wait (ms)
`Basic LRU-C`	1.56	0.18
+ `Dyn.-Batch-Write`	0.83	0.10
+ `LRU-D Mutex`	0	0.07

5.2 Evaluation Result

5.2.1 Baseline Performance. Let us first clarify the baseline performance of MySQL. DBMSs have many configurable knobs which can control their run-time behavior, and the knob-tuning can yield higher performance when properly tweaked [14, 41, 43, 44]. To investigate the effect of knob tuning on I/O serializations, we adjust key knobs critical to I/O performance and concurrency in InnoDB [31–33], and measure transaction throughput. For example, we tune the maximum IOPS used by the background flusher to exploit high IOPS of SSDs by adjusting knobs relevant to the I/O capacity such as `innodb_io_capacity`, `innodb_io_cpacity_max`, and `innodb_flush_neighbors`. A separate experiment indicates that the tuned MySQL outperforms default MySQL by 40% in terms of transaction throughput. Throughout this paper, we use the tuned MySQL as `Vanilla` for a fair comparison.

5.2.2 Performance Drill Down. In order to drill down the performance gains of the individual optimization technique and thus understand their contribution, we measure transaction throughput, read stall ratio, and buffer pool mutex contention while running the TPC-C benchmark with three different versions of LRU-C on `SSD-A`, and present the throughput results in Figure 6 and the remaining two metrics in Table 3. We calculate the read stall ratio by counting the number of single page flushes upon page miss and dividing it with the number of total page miss, while performing the benchmark. Also, we collect the total mutex wait time of threads provided by the performance schema of MySQL [11]. In each experiment, the buffer size is set as 10% of the initial database size. Three LRU-C versions include the basic LRU-C version, LRU-C with dynamic batch write, and LRU-C with dynamic batch write and parallel mutex manipulation. They are denoted as `Basic LRU-C`, `+Dynamic-Batch-Write`, and `+LRU-D Mutex`, respectively, in Figure 6 and Table 3. We analyze how the two optimization methods affect the evaluation metrics by adding two optimizations in order.

The basic LRU-C improves the throughput over the Vanilla MySQL by 1.63x mainly because it can eliminate most read stalls. To be specific, the ratio of read stalls has reduced drastically from 33% (as given in Figure 2) to 1.56% (as presented in Table 3). Though infrequent, however, the basic LRU-C still incurs read stalls, which



(a) TPC-C **(b) LinkBench**

Figure 7: Transaction Throughput: Two OLTP Workloads (SSD-A)

can limit the throughput considerably [2]. When the number of page misses momentarily spikes (e.g., as the working set changes), the clean pages in the mixed region can be quickly consumed and thus the LRU-C pointer can eventually meet the LRU head. In such cases, even the LRU-C method has to choose the dirty page at the tail of the LRU list as the victim, causing read stalls.

When the dynamic-batch-write optimization is applied, LRU-C can further reduce the read stall ratio to less than 1%. This in turn improves the transaction throughput by 33%. By flushing more dirty pages in the dirty region in batch, the LRU-C with dynamic batch write enabled can generate more free buffer frames faster than the basic LRU-C. Thus, the optimization technique enables to close the gap between the free buffer consumption rate by foreground processes and the free buffer production rate by the background flusher. Because of the contention for the `LRU-mutex`, however, these two threads still have to wait for each other's work to complete.

By introducing the `LRU-D` mutex, the parallel LRU-list manipulation optimization allows foreground processes and background flusher to access the mixed and dirty region of the LRU-list independently and thus to issue their I/O requests in parallel. As a result, the optimization can further improve the throughput by 40%.As shown in the third column of Table 3, the average mutex wait time is reduced by 27%. Moreover, read stalls are completely eliminated.

5.2.3 Run-time Performance. This section compares the performance of LRU-C with Vanilla, WAR [2], and CF-LRU [36] in terms of throughput, system utilization, latency, and hit ratio.

Throughput LRU-C can improve the performance mainly by parallelizing read and write requests that were once serialized due to read stalls and mutex conflicts. As shown in Figure 7a, LRU-C outperforms Vanilla by three folds across different buffer sizes ranging from 5% to 25%. While WAR outperforms Vanilla by two folds by avoiding read stalls, it still underperforms LRU-C considerably. This is because WAR still serializes I/Os due to the LRU mutex, whereas LRU-C is free from such mutex-induced serializations. First, as in the vanilla MySQL, a foreground process in WAR can experience RR-serialization upon page miss. That is, the process must wait for the LRU mutex while another process is scanning the LRU tail. Second and to be worse, once a foreground process does not find a clean victim from the LRU tail, it has to, in some cases, wait for the background flusher to empty TWB (Temporary Write Buffer) [2]. That is, it encounters another type of RW-serialization. These two factors can explain the throughput gap between WAR and LRU-C in Figure 7a.

Table 4: CF-LRU Performance: Varying the Scan Depth

`lru_scan_depth`	512	1,024	2,048	4,096	8,192	16,384
TPS	243	247	255	346	329	316
Read Stall Ratio (%)	34	33	31	5	2	1

Also, we compare LRU-C with CF-LRU (Clean-First LRU), a flash-aware buffer replacement policy [36]. To emulate CF-LRU in InnoDB, we modify the value of the `innodb_lru_scan_depth` parameter according to the equation that defines the window size of CF-LRU [36]. The parameter determines the number of pages to be scanned from the LRU tail when searching for a clean frame upon a page miss. We run the TPC-C in the CF-LRU mode by varying the buffer pool size from 5% to 25% and represent the corresponding TPS values in Figure 7a. The performance gains of CF-LRU over the default scan depth (i.e., 1,024) in all cases range between approximately 10% and 40%, a limited gain compared to WAR and LRU-C. To better understand the impact of the window size in CF-LRU on performance, we measured the transaction throughput while running the TPC-C benchmark on SSD-A with a 10% buffer pool size. We varied the scan depth for the 10% buffer size from the default value of 1,024 to 512, 2,048, 4,096, 8,192, and 16,384, respectively, and presented the results in Table 4. The transaction throughput peaks at a scan depth of 4,096 due to reduced read stalls but then declines with larger scan depths. This is because CF-LRU reduces read stalls, but at the expense of a longer scanning time for larger scan depths, which exacerbates the mutex-induced RR-serialization among foreground processes in need of free buffer frames. These results confirm that I/O serializations cannot be addressed solely by favoring clean pages as the victim over dirty ones.

To evaluate the effect of LRU-C on other workloads than TPC-C, we run the LinkBench workload on SSD-A with the same configuration as in the experiments in Figure 7a and present the result in Figure 7b. LRU-C processes twice more operations per second than the Vanilla, 78% more than CF-LRU, and 42% more than WAR, respectively. Given that the read-to-write ratios in the two workloads are almost the same [2], we expected that the gain in LinkBench would be similar to that in TPC-C. However, the performance gain in LinkBench is smaller than our expectation. This is partly because the LinkBench workload is more CPU-intensive than TPC-C; thus, the CPU power of the system used in the experiment limits the

Table 5: IOPS, Bandwidth, and CPU Utilization (SSD-A)

	IOPS (16KB)		Bandwidth (MB/s)			CPU Util.
	Read	Write	Read	Write	Total	(%)
Vanilla	6,860	3,968	109	62	171	24
WAR	13,887	6,336	222	99	321	79
LRU-C	25,510	7,424	408	116	524	84
FIO (R:W=2:1)	28,685	14,779	459	236	695	10
FIO (R:W=4:1)	39,755	11,222	636	179	815	9

Table 6: Transaction Latency (TPC-C, Buffer=10% SSD-A)

(unit: ms)	Avg Latency	95th Latency	99th Latency
Vanilla	4.04	332.16	583.05
WAR	1.95	380.83	754.27
LRU-C	1.32	53.01	391.26

Table 7: Miss Ratio Comparison

Buffer Size / DB Size (%)		5	10	15	20	25
Miss Ratio (%)	Vanilla	5.8	3.5	2.3	1.6	1.1
	LRU-C	7.0	4.4	3.1	1.9	1.2
LRU-C/ Vanilla		1.21	1.25	1.36	1.21	1.04

effect of LRU-C. Given the large performance disadvantage of CF-LRU compared to LRU-C and WAR on both TPC-C and LinkBench, we do not include CF-LRU in our remaining experiments.

CPU and I/O Utilization Table 5 summarizes the average value of read-write IOPS, I/O bandwidth, and CPU utilization measured while running the TPC-C benchmark on SSD-A with the buffer size of 10% in each of three MySQL modes. We obtain those metrics using the `iostat` utility provided in the Linux. As is shown in Table 5, all measured values of three MySQL modes are consistent with transaction throughput in Figure 7. Table 5 also presents the metrics obtained from running two synthetic I/O workloads.

Let us first discuss the benefits of LRU-C in terms of I/O metrics. First of all, LRU-C has higher write IOPS than Vanilla and WAR. This is because the dynamic-batch-write optimization flushes all the dirty victims left behind the LRU-C pointer, LRU-C can obviously achieve higher write throughput than WAR as well as Vanilla. Second, the read-write ratio for LRU-C (roughly 4:1) is notably higher than that of Vanilla and WAR (approximately 2:1). This is mainly due to the decreased hit ratio in LRU-C. As discussed in Section 5.2.3, the design of LRU-C trades hit ratio for higher I/O throughput, as with CF-LRU [36]. For instance, in the experimental setting for Table 5, the miss ratio in LRU-C is 25% higher than that in Vanilla (see the third column in Table 7), yielding more reads.

Table 5 indicates that LRU-C uses about three times more I/O bandwidth than Vanilla. It would also be meaningful to compare their bandwidths with the peak bandwidth of the device. To obtain the peak I/O performance of the device under the same read-write ratio of Vanilla and LRU-C, we measure the random read and write IOPS of 16KB pages on SSD-A using the FIO tool [4] with two read-write ratios of 2:1 and 4:1 and calculate the I/O bandwidths from the measured IOPSs. The results are presented as the last two rows in Table 5. Compared to the measured full bandwidths of SSD-A, LRU-C utilizes about 64% of the limit (i.e., 5th row), whereas Vanilla utilizes only about 25% of the limit (i.e., 4th row). This result confirms that LRU-C can leverage the I/O potential of the SSD device much better than WAR and Vanilla, although there is still some room for further optimization.

In terms of CPU utilization, as such, LRU-C is superior to Vanilla and WAR. Because I/O serializations no longer block foreground processes, CPU cores will be more active in LRU-C. In the case of Vanilla, once most foreground processes are read-stalled, CPU cores become idle. In addition, RR serialization remains unresolved even in the case of WAR without read stalls. As the foreground thread unnecessarily scans dirty pages repeatedly in the LRU tail for a clean page, it wastes the CPU cycle during the process while exacerbating mutex contention.

Transaction Latency The transaction tail latency holds significant importance as much as the transaction throughput because it can negatively impact user experience and client confidence [13, 18]. Hence, online service providers must proactively address tail latency to support service reliability and avoid monetary losses accordingly. LRU-C can reduce the transaction latency by eliminating the write latency from read stalls and mutex-induced serializations from the critical path of transactions.

To verify this effect, we measure average, 95th, and 99th transaction latencies while running the TPC-C benchmark in Vanilla, WAR, and LRU-C and present the results in Table 6. LRU-C considerably reduces the overall transaction latency and narrows the latency distribution: the average, 95th, and 99th percentile latency of LRU-C is 66%, 84%, and 33% lower than the Vanilla, respectively. LRU-C also outperforms WAR by far all in three latencies. In the case of WAR, its 95th and 99th latencies are even longer than Vanilla. This is because foreground read processes must wait for the background flusher to complete flushing all the dirty pages in TWB.

Trading Hit Ratio for Higher I/O Throughput LRU-C sacrifices a small proportion of hit ratio by selecting warmer clean pages as a victim upon page miss. Because of such victim selection method, LRU-C has a higher possibility to evict pages that are soon accessed in the near future, thus increasing the miss ratio [36]. To verify the effect of LRU-C on miss ratio, we measure the hit ratios while running the same TPC-C benchmark using Vanilla and LRU-C, and present their miss ratio (i.e., 1 - hit ratio) in Table 7. On average, the miss ratio of LRU-C is 20-30% larger than that of Vanilla. This partially accounts for why the read ratio of LRU-C is higher than Vanilla in Table 5. Although LRU-C compromises the hit ratio, its I/O performance gain from parallelized I/Os can far outweigh the reduction in the hit ratio.

5.2.4 Effects of Varying Read-Write Ratio. Recalling that the main goal of LRU-C is to make foreground read processes free from being stalled by I/O serializations, its effect is highly dependent on the relative ratio of reads and writes in the workloads. In particular, its impact is expected to reduce as the workload becomes read-intensive. To evaluate the effect of LRU-C under the varying read

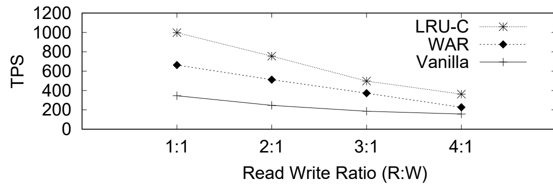


Figure 8: TPC-C Throughput: Varying Read-Write Ratio

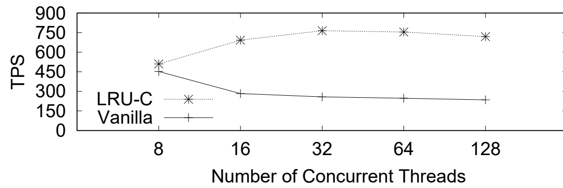


Figure 9: Impact of Concurrent Threads

and write (R:W) ratios at the block I/O layer, we run the TPC-C benchmark on SSD-A for four different R:W ratios, including 1:1, 2:1, 3:1, and 4:1, and the results are presented in Figure 8. In each experiment, the buffer size is set as 10% of the database size. The R:W ratio under the default transaction mix in the TPC-C workload is about 2:1. For other read/write ratios, we empirically adjusted the percents of five transaction types to achieve the desired read/write ratio. For example, for the R:W ratio of 4:1, the ratio of the read-only `stock` transactions is increased from 5% to 29% while that of the update-heavy `new-order` and `payment` transaction is decreased from 43% to 30%, respectively.

As expected, Figure 8 shows that the relative performance gain of LRU-C over Vanilla and WAR shrinks as the R:W ratio increases. This is mainly because read stalls decrease as the workload becomes read-intensive [2]. Nevertheless, in the read-intensive case of R:W = 4:1, LRU-C still outperforms Vanilla by two folds while WAR does only up to 35%. As the ratio of read stall diminishes in read-intensive workload, the effect of WAR also does. This is because the main role of WAR is to reduce read stalls. In contrast, LRU-C still improves throughput over Vanilla by two folds even in read-heavy workloads with less read stalls. This indicates that LRU-C is effective in mitigating mutex contentions as well as read stalls.

5.2.5 Effect of Multiple Concurrent Threads. To investigate the effect of LRU-C under different numbers of concurrent client threads, we measure the TPS (transactions per second) from running the same TPC-C benchmark used for Figure 7a using Vanilla MySQL and LRU-C, respectively, while varying the number of client threads from 8, 16, 32, 64, and 128. The experiment results are plotted in Figure 9, from which we make an intriguing observation. As the number of concurrent threads increases from eight to sixteen, the TPS of Vanilla halves while that of LRU-C increases by 50%.

With eight threads, the TPS of vanilla MySQL is relatively high. As the number of concurrent threads is identical to that of buffer pool instances, each thread can read and write to cached pages to a different buffer pool instance. Thus the mutex conflict between concurrent processes rarely occurs. Therefore, Vanilla does not suffer from I/O serializations with eight threads. For this reason, the TPS of LRU-C is almost the same as that of Vanilla. However, there are excessive concurrent users in real-world workloads [47]. Thus, in realistic cases (e.g., 64, 128 clients), the mutex contention partly causes the vanilla MySQL to halve its TPS. In addition, as the number of concurrent threads increases and thus multiple foreground threads consume the free buffer frame faster, foreground processes start encountering the read stalls.

In contrast, the TPS of LRU-C scales as the number of concurrent threads increases from eight to 32. LRU-C can be less effective with smaller number of concurrent threads because Vanilla is also free from mutex contention and read stall. However, the efficacy of LRU-C is prominent when the number of threads increases to 128. By reducing the mutex collision and minimizing I/O serializations, LRU-C improves the buffer cache's concurrency and scalability.

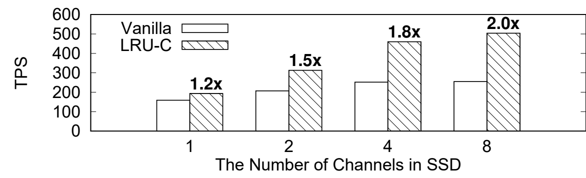


Figure 10: Impact of Device Parallelism

5.2.6 Effect of Device Parallelism. The multi-channel architecture of SSD provides high I/O parallelism. To see how LRU-C behave at different degrees of parallelism, we run the TPC-C benchmark the Cosmos+ OpenSSD [27] while reducing the number of channels in the board from eight to one. Even with the configuration of one channel, the way-level parallelism of eight still exists. We generate a 10GB TPC-C database (i.e., 100 warehouses) and use a 1GB buffer size (i.e., 10% of the database size).

As shown in Figure 10, LRU-C consistently outperforms Vanilla across all degrees of parallelism. In particular, as expected, as the degree of parallelism gets higher, the relative performance gap, as well as the absolute performance gap, are widening between the two. This result confirms that the parallelized I/Os in LRU-C will benefit more as more parallelism is available from the storage device. Thus, as the degree of parallelism in commercial SSDs is ever increasing, LRU-C becomes more beneficial accordingly.

5.2.7 Effects of Flash Device Type. To verify the impact of LRU-C on other SSDs with different I/O performances, we run the TPC-C benchmark on three other SSDs in Table 1. We also evaluate the effect of buffer size by varying it from 5% to 25% of the database size. The result in Figure 11 indicates that, regardless of SSD type, LRU-C outperforms Vanilla by at least 1.7x and up to 5.4x.

The effect of LRU-C varies depending on the I/O performance and parallelism in SSDs. In the case of SSD-B having similar IOPS capacity and architecture with SSD-A, the relative gap among the three schemes shows a similar trend with that in SSD-A. However, in the case of SSD-C with low IOPS, the relative performance gains of LRU-C and WAR over Vanilla are much larger than that in SSD-A and SSD-B. In particular, because of the slow write speed in SSD-C, Vanilla suffers from read stalls. In comparison, as shown in Figure 11c, the gain of LRU-C is relatively small because even Vanilla

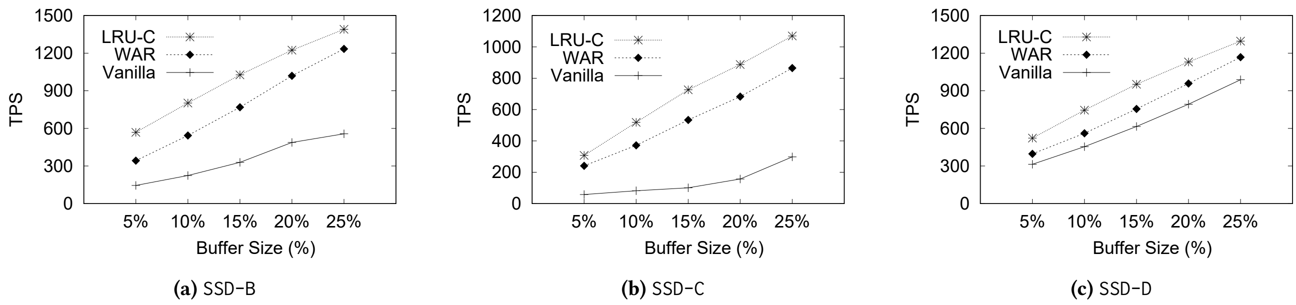


Figure 11: TPC-C Throughput: Three Different Flash SSDs

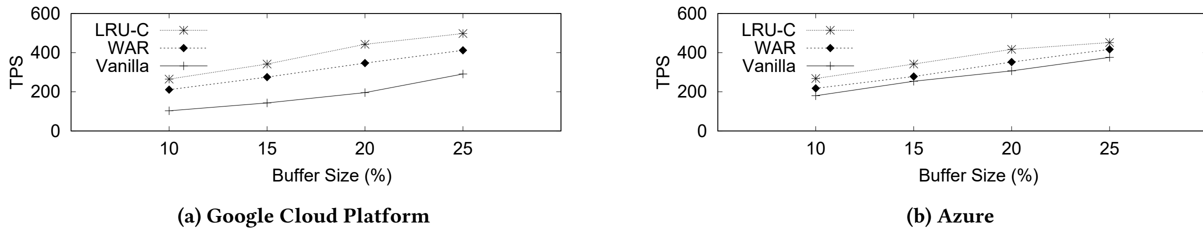


Figure 12: LRU-C on Cloud Storage

becomes CPU-bound on SSD-D. We conjecture that the performance gap is widened when tested on a system with more CPU power.

LRU-C on Cloud Storage To verify the effect of LRU-C on cloud storage, we also run the TPC-C benchmark on two cloud systems: GCP (Google Cloud Platform) [8] and Azure [30]. The storage options used in our cloud experiments are cloud block storage connected to compute instances over the regional data center network. For GCP, we use an n2-standard-8 compute engine with eight processors, 64GB memory, and a persistent SSD disk for database storage. For Azure, we use a standard F8s v3 compute engine with eight vCPUs, 16GB memory, and a persistent SSD for database storage. We create a 10GB TPC-C database and run the benchmark by changing the buffer size from 10% to 25% of the database size on both cloud platforms.

The experiment results in Figure 12 clearly indicate that LRU-C outperforms Vanilla and WAR also on both cloud platforms. According to Figure 12a, in GCP, LRU-C achieves 2.57x better throughput than Vanilla and 1.25x higher throughput than WAR. In Azure, LRU-C outperforms 1.49x and 1.23x than Vanilla and WAR, respectively, as shown in Figure 12b. The transaction throughput of Vanilla MySQL in Azure is higher than that in GCP because the cloud I/O performance of the former is better than that of the latter. In fact, we observed that the IOPS in Azure was higher than that in GCP while running Vanilla MySQL. Meanwhile, note that the throughputs of LRU-C are similar on both platforms. In fact, with LRU-C, both systems are CPU-bound (i.e., their CPU utilizations are more than 95%.) As such, the relative performance gain of LRU-C over Vanilla is less in Azure than in GCP. The performance gain of LRU-C is expected to be higher with more powerful CPU options. Though the network latency does exist on cloud platforms [28], the I/O wait time due to I/O serializations still impacts transaction throughput far more than the network latency [2]. Therefore, LRU-C is effective also on network-attached flash devices in cloud platforms.

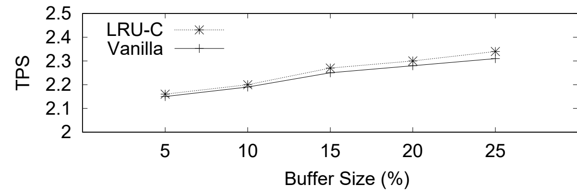


Figure 13: TPC-C Throughput on Hard Disk

Effects on Hard Disks For a comparison purpose, we measure the throughput of Vanilla, WAR, and LRU-C while running the same TPC-C benchmark used for Figure 11 on the hard disk from Table 1, and present the result in Figure 13. As expected, there is almost no performance gap between LRU-C and Vanilla. Because the hard disks have the same symmetric read speed and no parallelism, the Vanilla MySQL barely experiences the I/O serializations, so there is no room for performance improvement by the LRU-C method.

6 RELATED WORK

In that LRU-C is a flash-aware buffer management solution to avoid the I/O serializations and also to prioritize the read operations which are critical to transaction performance, several types of existing works are related: flash-aware buffer replacement [35, 36], read stall avoidance [2], prioritized read in flash devices [27, 34], flash-aware database systems [20, 29], database buffer management optimizations [5] and mutex optimization for buffer management [23].

Flash-aware Buffer Replacement Scheme Taking into account the imbalance of read and write costs in flash storage, a few flash-aware buffer replacement schemes including Clean-First LRU (CF-LRU) [36] have been proposed. Interestingly, the idea of favoring clean pages for eviction has been opted even for the hard disks [22]. LRU-C is common with CF-LRU because both prefer clean pages as victims, but they differ in several ways. First of all, LRU-C is not just a buffer replacement policy but a new buffer management architecture. Second, CF-LRU does not intend to address the I/O serializations. As demonstrated in Section 5.2.3, I/O serializations cannot be disentangled simply by taking the CF-LRU replacement policy. To address the problem more properly, LRU-C introduces the `LRU-C` pointer, and the `LRU-D` mutex provides two optimizations. Third, while any buffer manager taking the CF-LRU policy has to scan the same dirty pages in the LRU tail repeatedly, LRU-C need not have to do so with the help of the LRU-C pointer.

Read Stall Avoidance Protocol Recently, the WAR (write-after-read) protocol has been proposed as a new buffer management scheme to avoid the read stall problem [2]. It is common with LRU-C in that both addresses read stall problems. However, as shown in Section 5, LRU-C outperforms WAR considerably by solving the RR- and RW-serializations as well. Also, LRU-C has a few advantages over WAR, such as its simplicity in design and implementation. Unlike WAR requiring new memory resource, `TWB`, and several non-marginal algorithmic changes, LRU-C adds only two new data structures, the `LRU-C` pointer and the `LRU-D` mutex, to the existing list-based data structure and requires natural and moderate code extension for the existing buffer management module.

Prioritized Reads in Flash Storage Because the read operation is critical to the latency as well as the throughput of I/O-intensive applications, the flash storage controllers have introduced a few features which prioritize the read operations [1, 16, 27, 34]. One example is to suspend the preceding writes or even erase operations queued at a flash chip so as to serve the following read operations first, thus reducing the read latency [27]. Though effective in reducing read latency [46], the read prioritization technique is orthogonal to the I/O serialization problem originating from the host buffer cache layer [1]. That is, while the technique intends to resolve the interference between writes and reads at the channel level of the SSD, it is barely effective once the reads are stalled at the host buffer cache layer and therefore are not issued to the storage devices. Recently, as a new storage interface for flash storage, the fused read and write (RW) command is proposed to address the read stall problem by requesting both read and write requests with a single command to the storage [1]. The read request can be immediately served once the dirty page is copied to the storage buffer. Though quite novel and effective, it is a hardware-based solution. In contrast, LRU-C is purely software-based. Moreover, LRU-C addresses two other I/O serializations.

Flash-aware Database Systems A few proposals about flash-aware database systems have been made [20, 29]. Their main design objective is to minimize the I/O stack overhead on flash storage, not to address the I/O serialization problem. Because LeanStore [29] and DANA [20] are also based on the background writer, and the read latency is critical to the performance, the I/O serialization problem still persists in them. As such, they will benefit from LRU-C.

Buffer Management in Other DBMSs Like MySQL, Oracle also takes a variation of LRU to keep hot pages in the buffer cache. Oracle also scans for a clean page until designated scan depth upon page miss. But the difference between MySQL is that Oracle moves the dirty pages encountered during the scanning process to the LRU-W list [5]. Likewise, moving dirty pages in the LRU tail to a separate list removes the repetitive scanning process, which is common with LRU-C. However, the mutex overhead still exists when moving dirty buffer frames from the main LRU list to the LRU-W list. To be worse, when the LRU-W list becomes full, all the foreground processes have to wait until all dirty pages in the list are flushed, which can be considered as another form of read stall. In contrast, LRU-C is free from mutex contention and read stall.

Mutex Optimization for Scalable Buffer Management To our best knowledge, Shore-MT [23] is one of the first DBMS engines aiming at making buffer management scalable by addressing two types of mutex conflicts in the buffer manager. First, it reduces mutex contention inside the buffer pool by slicing the global mutex of a hash table so that a single mutex can protect a single hash bucket. Second, the buffer manager of Shore-MT using the clock algorithm reduces RR-serializations among reader processes while scanning the ring list for a victim page upon page miss.

With regard to mutex-related optimizations, LRU-C differs from Shore-MT in at least two ways. First, the LRU-C pointer enables each foreground process to get a clean frame faster and accordingly minimizes the *LRU-M* mutex holding time upon page miss. Second, introducing the *LRU-D* mutex allows the background flusher and foreground process to manipulate two regions of the LRU list concurrently and thus avoid the RW-serializations between them.

7 CONCLUSION

This paper elaborates on why the conventional database buffer managers cause the I/O serializations on flash storage with asymmetric read-write speed and high parallelism. To make the database I/Os parallel and thus leverage the parallelism in flash storage, we propose a simple but effective solution, LRU-C. By introducing the LRU-C pointer, which points to the first clean page from the LRU tail, LRU-C allows a page-missing foreground process to find a clean victim frame quickly and, more importantly, to avoid the read stall problem. The LRU-C pointer also enables the background writer to flush all dirty pages left behind the pointer in a batch, thus allowing higher write throughput. The introduction of the *LRU-C* pointer dissects the LRU list into two regions, mixed and dirty regions. Hence, by introducing an additional mutex, `LRU-D`, LRU-C parallelizes database I/Os. In these ways, LRU-C can fully leverage the internal parallelism in flash storage, thereby enhancing CPU and I/O utilization accordingly.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2022R1A2C2008225) and by Samsung Electronics. We are thankful to the anonymous reviewers for their insightful feedback.

REFERENCES

[1] M. An, S. Im, D. Jung, and S.-W. Lee. Your read is our priority in flash storage. *Proc. VLDB Endow.*, 15(9):1911–1923, 2022.

[2] M. An, I.-Y. Song, Y.-H. Song, and S.-W. Lee. Avoiding read stalls on flash storage. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1404–1417, New York, NY, USA, 2022. Association for Computing Machinery.

[3] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. 2013.

[4] J. Axboe. Fio (flexible io tester). https://github.com/axboe/fio, 2022.

[5] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. R. Loaiza, and N. MacNaughton. The oracle universal server buffer manager. 1998.

[6] F. Chen, B. Hou, and R. Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Trans. Storage*, 12(3), may 2016.

[7] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, 2011.

[8] G. cloud platform. Google cloud. https://cloud.google.com/, 2022.

[9] M. T. M. T. O. Corp). Buffer pool. https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html, 2022.

[10] M. T. O. Corp). Innodb checkpoints. https://dev.mysql.com/doc/refman/5.7/en/innodb-checkpoints.html, 2022.

[11] M. T. O. Corp). Monitoring innodb mutex waits using performance schema. https://dev.mysql.com/doc/refman/8.0/en/monitor-innodb-mutex-waits-performance-schema.html, 2022.

[12] M. T. O. Corp). Mysql 8.0 reference manual. https://dev.mysql.com/doc/refman/8.0/en/, 2022.

[13] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[14] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, 2005.

[15] J. Do, I. L. Picoli, D. Lomet, and P. Bonnet. Better database cost/performance via batched i/o on programmable ssd. *The VLDB Journal*, 30(3):403–424, may 2021.

[16] N. Elyasi, C. Choi, A. Sivasubramaniam, J. Yang, and V. Balakrishnan. Trimming the tail for deterministic read performance in ssds. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 49–58, 2019.

[17] facebookarchive. Linkbench. https://github.com/facebookarchive/linkbench, 2017.

[18] M. Fruth, S. Scherzinger, W. Mauerer, and R. Ramsauer. *Tell-Tale Tail Latencies: Pitfalls and Perils in Database Benchmarking*, pages 119–134. 01 2022.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[20] G. Haas, M. Haubenschild, and V. Leis. Exploiting directly-attached nvme arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020*, 2020.

[21] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 981–992, New York, NY, USA, 2008. Association for Computing Machinery.

[22] R. Jauhari, M. J. Carey, and M. Livny. Priority-hints: An algorithm for priority-based buffer management. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, page 708–721, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[23] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, page 24–35, New York, NY, USA, 2009. Association for Computing Machinery.

[24] A. Joshi, W. Bridge, J. R. Loaiza, and T. Lahiri. Checkpointing in oracle. In *VLDB*, 1998.

[25] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535, 2014.

[26] M. Jung, E. H. Wilson, and M. Kandemir. Physically addressed queueing (paq): Improving parallelism in solid state disks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2012.

[27] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. Song. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Transactions on Storage*, 16, 05 2020.

[28] C. Labs. 2022 cloud report. https://www.cockroachlabs.com/guides/2022-cloud-report/#form, 2022.

[29] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[30] Microsoft. Azure. https://azure.microsoft.com/en-us/, 2022.

[31] MySQL Team (Oracle Corp.). Configuring buffer pool flushing. https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-flushing.html, 2023.

[32] MySQL Team (Oracle Corp.). Optimizing innodb disk i/o. https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-diskio.html, 2023.

[33] MySQL Team (Oracle Corp.). Server system variable reference. https://dev.mysql.com/doc/refman/5.7/en/server-system-variable-reference.html, 2023.

[34] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, 60(5):653–666, 2011.

[35] S. T. On, S. Gao, B. He, M. Wu, Q. Luo, and J. Xu. Fd-buffer: A cost-based adaptive buffer replacement algorithm for flashmemory devices. *IEEE Transactions on Computers*, 63(9):2288–2301, 2014.

[36] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee. Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 234–241, New York, NY, USA, 2006. Association for Computing Machinery.

[37] Percona. tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql, 2018.

[38] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, Inc., USA, 3 edition, 2002.

[39] E. Rogov. Wal in postgresql: 3. checkpoint. https://postgrespro.com/blog/pgsql/5967965, 2022.

[40] P. Server. Xtradb performance improvements for i/o-bound highly-concurrent workloads. https://www.percona.com/doc/percona-server/5.6/performance/xtradb_performance_improvements_for_io-bound_highly-concurrent_workloads.html#xtradb-performance-improvements-for-io-bound-highly-concurrent-workloads, 2022.

[41] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1081–1092. VLDB Endowment, 2006.

[42] J. Z. Teng and R. A. Gumaer. Managing ibm database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[43] The PostgreSQL Global Development Group. Postgresql 11 documentation: Resource consumption. https://www.postgresql.org/docs/current/runtime-config-resource.html, 2019.

[44] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[45] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 10, USA, 2012. USENIX Association.

[46] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 10, USA, 2012. USENIX Association.

[47] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. 8(3):209–220, nov 2014.