Your Read is Our Priority in Flash Storage

Mijin An* Soojun Im** Dawoon Jung** Sang-Won Lee*
*Sungkyunkwan University
Suwon, Korea
{meeeejin,swlee}@skku.edu
**Samsung Electronics Co.
Hwasung, Korea
{soojun.im,dw0904.jung}@samsung.com

ABSTRACT

When replacing a dirty victim page upon page miss, the conventional buffer managers flush the dirty victim first to the storage before reading the missing page. This *read-after-write* (RAW) protocol, unfortunately, causes the *read stall* problem on flash storage; because of the asymmetric I/O speed and parallelism in flash storage, the clean frames are quickly consumed, so the read for the missing page often has to wait for the slow write to complete and for the frame to be clean due to the *resource conflict* for the same buffer frame. RAW will thus make the performance-critical synchronous reads often blocked by writes, severely worsening transaction throughput and latency. In addition, its strict I/O ordering will make flash storage with abundant parallelism under-utilized.

To avoid read stalls in the DBMS buffer, we propose RW (*fused read and write*) as a new storage interface. Using RW on read stall, the buffer manager can issue both read and write requests at once to the storage. Then, once the dirty page is copied to the storage buffer, it can immediately serve the read. In addition, to resolve read stalls in the flash storage buffer, we propose R-Buf, where the read buffer is separated from the write buffer so that reads can proceed at no stall. RW and R-Buf, working at different layers, complement each other when used together. We prototype RW and R-Buf on a real Cosmos+ OpenSSD board. Evaluation results show that RW alone improves TPC-C throughput over RAW by 3.2x and, combined with R-Buf, does by 3.9x. In addition, we demonstrate that R-Buf effectively mitigates the I/O interference in multi-tenancy.

PVLDB Reference Format:
Mijin An, Soojun Im, Dawoon Jung, Sang-Won Lee. Your Read is Our Priority in Flash Storage. PVLDB, 15(9): 1911 - 1923, 2022.
doi:10.14778/3538598.3538612

PVLDB Artifact Availability:
The source code is available at https://github.com/meeeejin/rw-rbuf.

1 INTRODUCTION

The buffer manager is at the core of database management systems because it includes both replacement policy and interaction with storage, which are critical to high performance. One common scheme closely involved with this is *read-after-write* (RAW). On a page miss, the buffer manager will choose a victim page based on its replacement policy. If the victim is dirty, it will *first write* the victim to the storage to clean the frame, and *then read* the missing page from the storage into it. In this situation, each page-missed process may suffer from the *read stall* problem because two I/O operations share one buffer frame: it contains the dirty page to write, and at the same time, the missing page will be fetched into it. However, the dirty page happens to be chosen as a victim according to the replacement policy and no other reason except for the resource conflict exists to follow the strict ordering between write and read.

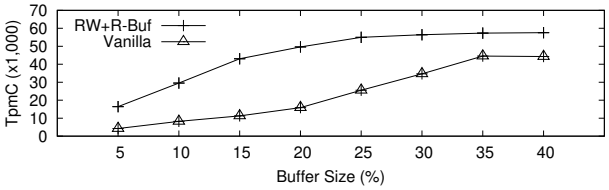


Figure 1: TPC-C Throughput: Vanilla vs. Proposed Scheme

Meanwhile, the all-flash era has arrived in all areas of computing. In particular, due to the superiority of flash storage in terms of random IOPS/$ and power consumption [12], the database market uses SSDs as the primary storage instead of hard disks. These SSDs have one inherent characteristic distinguishable from hard disks — *the asymmetry of read and write speed*. Also, they come up with abundant internal parallelism. Thus, their read IOPS are higher than the write IOPS.

Then, how does this asymmetry affect buffer management when running OLTP databases on flash storage? An obvious implication is that the rate at which the background writers produce free frames lags far behind the rate at which the foreground processes consume them. In RAW, even asynchronous batch-oriented background writers can hardly hide the slow write speed, so page replacements often involve synchronous writes [14]. As a result, foreground processes will experience frequent *read stalls* on SSDs.

Moreover, read stalls can also occur *inside SSDs* where the data buffer is shared by read and write requests. In a shared buffer, when no clean buffer frame is available, a read request from the host has to wait for the frame keeping a dirty page to be flushed to the flash chip. Thus, the host process with a missing page can experience even two stalls in series while reading the page: one at the DBMS buffer and the other at the storage buffer. Although synchronous reads are on the critical path of database applications, the RAW protocol taken by both buffers and the resulting read stalls will prolong the read latency. This causes host processes to be idle longer, worsening their throughput and latency. In addition, the strict *write-then-read* serialization by RAW will under-utilize the parallelism in flash storage. As shown in Figure 1, when running the TPC-C benchmark using the existing RAW protocol for both buffers, the transaction throughput (denoted as `Vanilla`) is consistently lower than the potential (denoted as `RW+R-Buf`) across various buffer sizes.

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538612

To avoid read stalls in the DBMS buffer, we first propose RW (*fused read and write*) as a new storage interface for flash storage. Upon read stalls, RW allows host applications to issue both read and write requests via one I/O command to the storage. Upon receiving the RW command, flash storage prioritizes read while handling read and write in parallel: the read is served immediately once the dirty page is copied to the storage buffer. In addition, to resolve the interference between concurrent reads and writes at flash storage buffer, we suggest R-Buf, a simple but effective *read-dedicated buffer* architecture. In R-Buf, since the read buffer is separated from the write buffer, read requests from the host can be handled faster without interference with write requests. With the help of RW and R-Buf prioritizing reads, slow writes are removed from the critical path of host applications. This improve their throughput and latency significantly. In addition, allowing write and read to be issued in parallel to the flash storage and flash channels/ways, RW and R-Buf can better utilize the internal parallelism of flash storage [6]. The main contributions of this paper are as follows:

- We identify the RAW protocol as an intrinsic impediment to the performance of transactional databases running on flash storage. We also detail how RAW can make processes vulnerable to read stalls at both host and storage buffer tiers on flash storage with the I/O asymmetry.
- To address read stalls at the host and storage buffers, we propose two solutions for flash storage: RW and R-Buf. They complement each other when used together.
- We prototype RW and R-Buf on a real Cosmos+ OpenSSD board by extending its firmware code. Experimental results show that RW alone outperforms RAW by 3.2x in terms of throughput and, combined with R-Buf, does by 3.9x.
- We also show that R-Buf is effective in mitigating the I/O interference in multi-tenancy. R-Buf can make two tenants perform more *proportional* than S-Buf, exploiting the internal parallelism of flash storage better.

2 BACKGROUND

This section explains the intrinsic I/O asymmetry of flash SSDs and describes the RAW protocol in the DBMS and storage buffers.

2.1 I/O Asymmetry in Flash SSDs

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. For example, in the case of MLC flash memory chips, a page write against a clean block takes 1,500us while a page read does only 50us [26]. In addition, the costly but inevitable garbage collection operations further widen the gap between read and write speeds at the device level.

To evaluate the asymmetry ratio of today's commercial SSDs, we use the synthetic I/O benchmark tool FIO [3] to measure the 4KB random read and write IOPS of four SSDs and one OpenSSD [21]. Each IOPS is measured with a queue depth of 32 for six hours on each SSD half-filled with data. Results are summarized in Table 1. For comparison, an enterprise-class hard disk is also presented in the table. The asymmetric ratio is given in the last column of the table, calculated by dividing the read IOPS by the write IOPS.

Table 1: Read and Write IOPS

Storage Media	Capacity (GB)	Random IOPS (4KB)		Asym. Ratio (R/W)
		Read	Write	
SSD-A†	1,024	529,977	51,077	10.4
SSD-B‡	400	264,846	19,705	13.4
SSD-C¶	1,024	723,241	70,983	10.2
SSD-D♯	250	99,123	8,739	11.3
OpenSSD†	32	142,032	16,355	8.7
HDD§	1,024	1,509	1,420	1.1

†Samsung 970 PRO NVMe, ‡Intel DC P3600, ¶Intel DC P4510, ♯Micron Crucial MX500, †Cosmos+ OpenSSD, §WD WD10EZEX

As shown in Table 1, the read IOPS of flash SSDs is at least eight times higher than the write IOPS. In particular for SSD-B, its read IOPS is higher than its write IOPS by more than thirteen-fold. Contrariwise, the random read and write IOPS of the hard disk are almost the same. To summarize, the asymmetry ratio varies with the individual flash storage and over-provisioning capacity, but the asymmetry is an inherent characteristic of flash storage [18, 26, 33].

2.2 RAW Protocol in DBMS Buffer

On a page miss, if no buffer frame is available, the page-missed process has to obtain a frame by *first writing* the dirty page to the storage and *then read* the missing page into the cleaned frame. We call this *strict write-then-read ordering* in replacing dirty pages from the buffer cache as the *read-after-write* (RAW) protocol. In this situation, the read operation is blocked by a write operation, which we call a *read stall*. This paper assumes LRU as the buffer replacement policy for simplicity of discussion. However, the problem of RAW is equally applicable to other policies without loss of generality.

Meanwhile, given that the database durability is guaranteed by forcing redo logs upon commit and that many database engines follow the *no-force* policy in their buffer management [13], writes in the database are asynchronous in nature. Therefore, major database engines, including Oracle, IBM DB2, and MySQL, are equipped with the background writers [4, 28, 39]. The background writer aims to produce free frames in advance by pre-flushing dirty pages so that each page-missed foreground process can immediately obtain a free frame from the free list.

However, the background writer lags behind in pre-flushing dirty pages on flash storage due to its asymmetric I/O speeds and high internal parallelism. In other words, since SSD shows fast read speed and its multi-level parallelism improves read processing, free frames in the free list are quickly consumed by foreground processes. For the same reason, the clean frames at the LRU tail are also promptly exhausted. As a result, the speed of producing available buffer frames cannot keep up with the speed of consuming them on top of read-fast but write-slow SSDs. This leads to the situation where foreground processes start experiencing undesirable read stalls [14]. As will be discussed in Section 5.1.1, the background writer still falls behind in producing free frames in RAW, despite the best endeavor to make it flush dirty pages actively by adjusting the flash-aware configuration knobs.

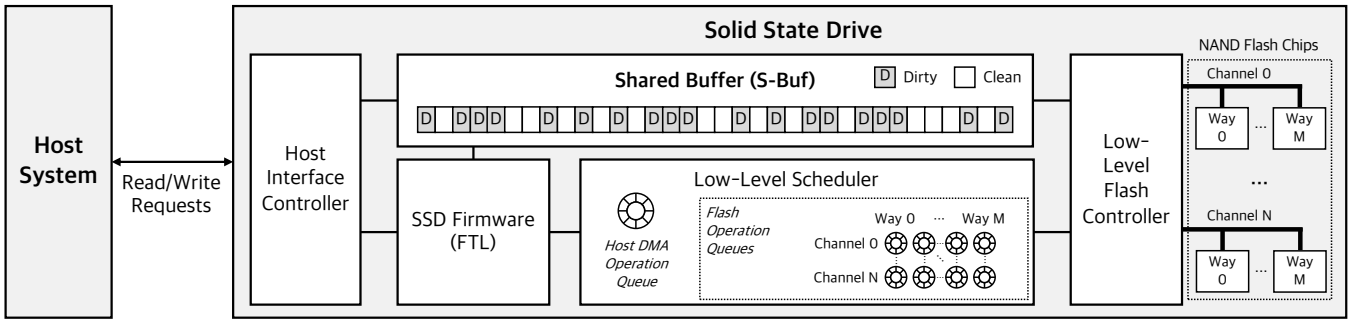


Figure 2: SSD Architecture and RAW Protocol in Storage Buffer

Table 2: Read Stalls in RAW Protocol (TPC-C on MySQL)

	Hard Disk	Flash SSDs				
	(HDD)	SSD-A	SSD-B	SSD-C	SSD-D	OpenSSD
Read Stalls	0%	28%	28%	36%	33%	32%

(a) Varying Storage Devices (Buffer Size = 10%)

Buffer Size	10%	20%	30%	40%	50%	60%
Hit Ratio	92.6%	97.2%	97.6%	98.6%	98.8%	99.3%
Read Stalls	28%	32%	32%	31%	16%	0%

(b) Varying Buffer Size (SSD-B)

2.3 RAW Protocol in Storage Buffer

Flash storage devices also have a DRAM area used as a data buffer for data transfer between the host system and the flash chips. In this paper, the data buffer is referred to as S-Buf to emphasize that it is shared for both read and write requests, thus distinguishing it from our approach, R-Buf. Since there is no publicly available data about the internal architecture and implementation details of commercial SSDs, we describe the internal I/O handling process of SSDs based on the recent research prototypes [21, 31]. S-Buf is where the data to write is temporarily stored before being transferred to NAND flash chips. It is also where data read from the flash chip is stored before being sent to the host. Its main goal is to reduce the performance gap between the host system and the physical device by temporarily holding the I/O data. In particular, it tries to hide the long latency of flash operations for writes by absorbing the write data in S-Buf [21].

Now let us explain how read requests are handled inside SSD using Figure 2. When the host sends a read command, the host interface controller interprets the command and forwards it to the FTL. Then, the FTL generates a set of subsequent operations (*e.g.*, DMA and flash operations) to service it and tries to allocate a data buffer frame from S-Buf to store the read data. For this, a victim frame is selected according to the replacement policy (*e.g.*, LRU). If the victim frame is clean, its content is simply discarded, but if dirty, its content should be written to the NAND flash chip before using it. That is, the read command should wait for the NAND write to complete; thus, the read stall occurs in S-Buf. After securing a free buffer frame through this process, the read operation is sent to the corresponding flash operation queue. Then, the low-level scheduler compares the priority of each operation in the queue and issues the read operation to the NAND flash chips when ready. When the NAND read completes, the read data is transferred to the clean buffer frame through the flash DMA (direct memory access) operation, and then it is finally transferred to the host memory through the host DMA operation. In summary, the storage buffer is also taking the RAW protocol, which can cause read stalls inside SSDs.

3 READ STALL IN DBMS AND RW COMMAND

This section details why RAW in DBMS causes significant read stalls on SSDs and demonstrates how serious the problem is in a realistic workload. Then, as an explicit way to address read stalls in the DBMS buffer, we propose a new storage command, RW.

3.1 Read Stalls in Relational DBMS Buffer

To check the severity of read stalls in real database systems, we measure the ratio of read stalls out of all read requests while running the TPC-C benchmark [24] on MySQL. In the experiment, the buffer size is set to 10% of the initial database size. The same experiment is repeated for each of five SSDs and one hard disk in Table 1. The results are summarized in Table 2a.

In hard disk, all foreground processes obtain their free frames without stalls. Due to its symmetric read and write speeds, the rate at which the background writers produce free frames is balanced with the rate at which foreground processes consume them.

On the other hand, all the five SSDs consistently suffer from excessive read stalls. To be specific, more than one fourth of the page-missed processes have to wait for the slow writes to complete and free frames to be secured. This is because the read IOPS is at least eight times higher than the write IOPS in flash storage, so even if the background writers do their best to actively flush dirty pages, free frames are promptly exhausted by foreground processes. As outlined in Figure 1 and will be detailed in Section 5.3, such read stalls severely limit the transaction throughput.

Meanwhile, the read stall problem can be alleviated or even disappear with a large buffer, as fewer reads means fewer pending writes. Hence, we measure the hit ratio and read stall ratio while

running the TPC-C benchmark on SSD-B by changing the buffer size from 10% to 60% of the database size. The results are presented in Table 2b. As the buffer size increases, the hit ratio also does from 92% to 99%, and read IOPS accordingly drops. However, read stalls remain consistently high until the buffer size becomes as large as 40% of the database size. It drops to zero when the buffer size reaches to 60%. For this reason, as illustrated in Figure 1 and will be detailed in Section 5.3, the performance of RAW is suboptimal over a wide range of buffer sizes.

In addition, to confirm the read stall problem in other DBMSs, separate experiments are performed with the same configuration used in Table 2a using PostgreSQL and Oracle on SSD-A. Experimental results show that the read stall ratios are 36% in Oracle and 61% in PostgreSQL. Since both DBMSs use the RAW protocol upon page miss [1, 4], read stalls are unavoidable for them. In summary, the degree of read stalls varies depending on the DBMS engine, but the problem is inherent in RAW-based DBMSs running on SSDs.

3.1.1 Problem Definition. When a page-missed process falls in a read stall, a missing page P_R can be read into a buffer frame f only after the dirty victim P_W occupying f is completely flushed and f becomes clean. In this respect, the buffer frame f is a *shared resource* between the write for P_W and the read for P_R. Thus, two I/O operations have to be performed *in serial*: the read has to wait for the preceding write to complete. Except for this, no other reason exists to follow the strict ordering between the two. That is, P_W *happens to* be selected as a victim based on the replacement policy. In summary, the *resource conflict* in RAW forces two unrelated read and write to be serially requested to the storage, causing the read stall problem. The write-then-read serialization at the database buffer layer, in turn, blocks the opportunities to issue read and write requests in parallel using the low-level asynchronous I/O primitives (*e.g.*, `libaio` or `io_uring`), combine them in the storage command queue (*e.g.*, native command queue) and process them in parallel inside flash storage [6].

Now let us discuss how read stalls affect system utilization and transaction performance. First, on a read stall, the page-missed process becomes idle simply waiting for the previous write to complete, degrading CPU utilization. Secondly, the strict I/O ordering prevents the system from taking advantage of the parallel support of SSDs, lowering storage utilization. Lastly, read stalls increase transaction latency. In principle, the synchronous read should be in the critical path of the ongoing transaction, but the write is not in the path in the ideal case where dirty pages are asynchronously pre-flushed in advance. However, when the transaction encounters a read stall, the time taken to finish the write will be added to the critical path of the transaction, prolonging its total latency.

3.2 RW Command

3.2.1 Key Idea. As mentioned in Section 3.1.1, the read stall problem is due to the resource conflict: write and read requests that happen to share one buffer frame on a page miss must be executed in serial rather than parallel. Therefore, the inevitable I/O serialization at the host buffer cannot fully utilize the parallelism of SSDs. Meanwhile, recalling that the buffer manager knows the target page addresses of both write and read requests on a read stall, two I/O operations can proceed in parallel if they satisfy the following condition: read a new page into the shared buffer frame only after the dirty page copy is flushed to the storage buffer. If any I/O command with this semantic does exist, the read stall problem can be avoided. Unfortunately, the existing storage interface only provides separate read and write commands. This recognition leads us to propose a new block I/O command, RW (fused Read and Write), which is used to write a dirty page to the storage and, in parallel, read the missing page to the host in one I/O call.

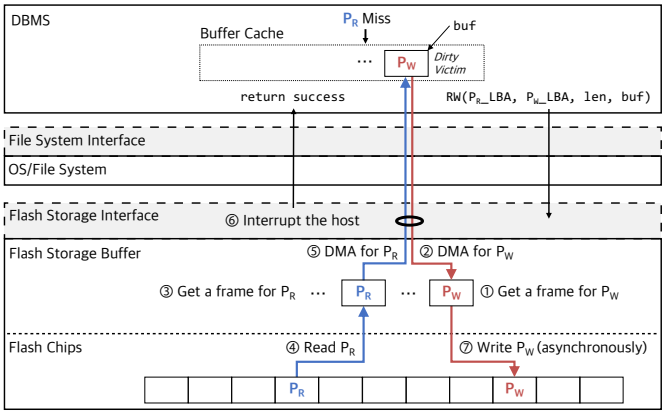


Figure 3: How RW Works: An Illustration

3.2.2 Abstraction and Architecture. Since no matching command exists in current storage interface, RW is added as an NVMe vendor specific command. Its parameters and semantic are as follows:

rw(rdLBA, wrLBA, len, buf) The first and second parameters, `rdLBA` and `wrLBA`, are logical block addresses of two data pages to read and write, respectively. The third parameter, `len`, indicates the length of two pages: their sizes are assumed to be same. The fourth parameter, `buf`, is the starting virtual address of the host buffer space that holds the dirty page to write and, at the same time, the destination to read a new page. When the host issues an RW command, both write and read requests are handled in parallel by the NVMe controller. Once the dirty page with `wrLBA` is copied to the storage buffer, and thus the host buffer frame becomes reusable, the page with `rdLBA` can be read into that host buffer frame. Therefore, the read operation can proceed without interference from the write operation for the dirty victim, so it can avoid read stalls. Note that RW exploits in-storage buffer frames to resolve the resource conflict between write and read operations.

Figure 3 overviews how RW works when a page P_R is missing and a dirty page P_W becomes the victim for the replacement. Upon receiving RW with two LBAs (P_R_LBA for read and P_W_LBA for write) and a host buffer address (`buf`), the FTL first tries to obtain a buffer frame for write from its storage buffer (①). It then performs a DMA to copy the host data pointed by `buf` to the obtained frame (②). Next, after securing another buffer frame for read (③), the FTL reads the page of P_R_LBA from the flash chip into the frame (④) and then performs the second DMA to transfer the page to the

host (⑤). The storage controller then sends an interrupt to the host to indicate that RW completes successfully (⑥). Meanwhile, the data of P_W_LBA is asynchronously written to the flash chip (⑦).

Consistency and Durability In very rare cases, successive RW calls to the same page can occur. To always return the latest version of the page, there is a command queue on the host interface controller. SSDs have a local host command queue to store I/O requests from the host system. Host requests are put into this queue and fetched in order [21, 31]. RW follows this existing command queue semantic to ensure consistency, so the out-of-order problem does not occur. RW also does not peril the durability of the I/O interface in the existing block device. That is, the durability requirement for writes in the RW command is simply the same as that of the existing write command. In fact, as mentioned in Section 2.2, the write operation in the database is asynchronous in nature, and the storage engine does not assume that the durability for the existing write operation is guaranteed. Likewise, the RW command assumes that the durability for the write operation is not guaranteed, but the failed write is recoverable using the redo log.

Benefits To our best knowledge, RW is the first storage interface that carries out two I/O operations in one call. This simple feature will bring advantages in resolving the read stall problem.

First of all, RW is an intuitive and direct solution to address the read stall problem because read and write requests are not serialized but issued together at once, and the read request is serviced at the earliest time. It will also make the parallelism in flash SSDs better utilized. Second, RW will simplify the logic of the buffer manager. A buffer manager suffices to simply invoke RW upon read stall without resorting to any complicated software-based mechanism in RAW. In other words, the process of writing the dirty victim to storage, emptying the buffer frame, and then reading the missing page into the frame is replaced by one RW call. Third, the simplified buffer manager logic will mitigate the run-time overhead due to buffer replacement. In RAW, a buffer frame holding the dirty victim page has to move from the LRU tail to the free page list and, after reading the missing page, has to move back to the LRU head. This process will incur four mutex acquisitions and four list modification operations: two for the LRU list and two for the free list. In contrast, RW will require only two mutex acquisitions and two list modifications for the LRU list since it can eliminate CPU works such as free block acquisition. Lastly, RW reduces the kernel I/O stack overheads. It is well known that one I/O call requires two context switches and one interrupt [38]. Therefore, by replacing two I/O calls for write and read with one call, RW can reduce the number of I/O interrupts and the number of context switches accordingly. In addition, recalling that these kernel I/O stack overheads are non-marginal on top of flash storage with low latency [38, 43, 45], the reduced I/O calls will reduce the CPU instructions consumed by the kernel I/O stack. Though we illustrate its benefit using the database buffer, RW will also be beneficial to the OS buffer cache with the RAW protocol (*e.g.*, Linux page cache with 2Q algorithm).

RW Abstraction for Multiple Devices The RW command assumes that the write and read pages are on the same device. This assumption may not be valid in large databases spanning multiple storage devices where read and write pages are located in different physical devices. Fortunately, however, those devices are usually exposed to the database layer as one logical volume. Hence, the effect of the RW command remains valid in such scenarios once RAID-like solutions support the abstraction.

3.2.3 Prototype Implementation. We prototype RW by adding it as a new NVMe command to an OpenSSD board and extending its firmware code to support the semantic. In addition, to evaluate its performance, we also modify the MySQL/InnoDB engine to call RW on read stalls.

Changes made in OpenSSD An OpenSSD (open-source SSD) platform allows to modify its hardware and software design freely [16]. To implement RW and evaluate its performance, we use a real OpenSSD, *Cosmos+ board* [21], which supports the NVMe interface. We extend the NVM command set by defining an opcode for RW and implementing a custom operation in the firmware. When the host calls RW with the specified opcode, the host interface controller identifies the given NVM command set and prepares each read and write. Since addresses to read and write differ from each other, the NAND device controller can perform two operations in parallel utilizing multiple channels inside the SSD.

Changes made in MySQL To leverage RW on MySQL/InnoDB, we use the `ioctl` system call and modify the buffer manager of MySQL. Since RW is not always available for MySQL and other applications that access files through a file system, we extend `ioctl` to allow the host to send the NVM command set for RW. Thus, RW can pass through the file system to the storage device instead of invoking the NVMe command directly from applications.

For MySQL, we modify the buffer manager (`buf`) and file I/O (`fil`) modules. First, we newly implement an I/O function so that the buffer manager can calculate LBAs for read and write and issue RW to the storage. We also modify the read function of MySQL so that, after RW returns, the transaction can proceed immediately using the read data buffer without a separate free buffer acquisition procedure. At this point, the read data buffer is moved to the head of the LRU list according to the LRU policy. The code changes made in MySQL/InnoDB are minimal, which is less than 130 lines of code in mainly two modules: `buf` and `fil`. In addition, note that, on read stall, due to its simplified buffer manager logic, RW can actually run with at least 840 fewer lines of code than RAW.

4 READ STALL IN STORAGE AND R-BUF

Even when a read request can be made from the host without stall with the help of RW, it could unfortunately encounter another stall in the flash storage buffer for the same reason (*i.e.*, resource conflict). This section illustrates the impact of the RAW protocol taken by the storage buffer in the existing SSDs on read performance and then suggests R-Buf to address read stalls in flash storage.

4.1 Read Stalls in SSD Buffer

First, to check the impact of read stalls on read latency inside flash storage, we measure the average read latency using the FIO tool. We run two random I/O workloads, `randread` and `randrw`, separately for each of the three commercial SSDs in Table 1. For `Read-Only` (`randread`), we run four FIO threads each of which issues random reads of 4KB pages with the queue depth of 32. For `Reads+Writes`

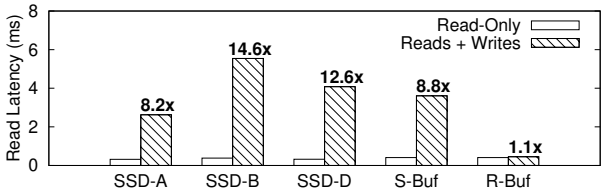


Figure 4: Effect of Concurrent Writes on Read Latency (FIO)

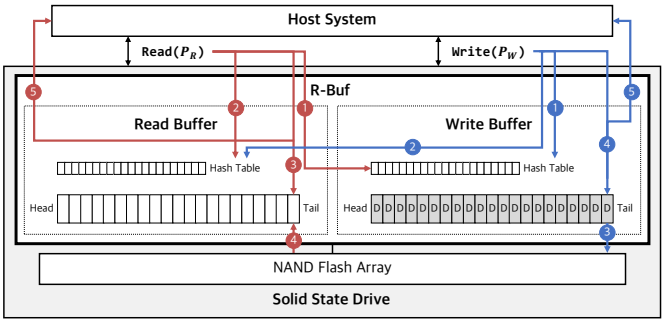


Figure 5: R-Buf in SSD

(`randrw`), we run two threads for reads and the other two for writes. In both cases, direct I/O is used to bypass the page cache of the kernel. Because we cannot directly access the firmware of commercial SSDs, we measure the read latency at the block I/O layer rather than inside storage, using the `blktrace` utility in Linux. For comparison, we also conduct the same experiment using Cosmos+ OpenSSD in two different buffer schemes: S-Buf and our proposed technique R-Buf. The results are presented in Figure 4.

As shown in Figure 4, the latency of reads when mixed with writes (`Reads+Writes`) increases significantly compared to that of `Read-Only` across all commercial SSDs and Cosmos+ OpenSSD with S-Buf (*e.g.*, by up to 14.6 in `SSD-B`). In contrast, in the case of Cosmos+ OpenSSD with R-Buf, the read latency is nearly unaffected by the concurrent writes. This is because in R-Buf, as detailed later, write requests will not interfere with read requests.

4.1.1 Problem Definition. In S-Buf, when the victim buffer into which data will be read is dirty, the read operation should wait until the dirty victim is written to NAND. Since S-Buf holds both read and write requests from the host and follows the RAW protocol, reads are frequently blocked by the preceding dirty victim writes. Further, when the preceding write triggers the costlier garbage collection operation, the stalled read will be blocked much longer. Therefore, one flash write (and a garbage collection in some cases) will be added to the latency of stalled read requests. In addition to prolonging the latency of read requests, the RAW protocol serializes write and read requests to the underlying flash controller, which makes the abundant parallelism in flash storage under-utilized.

In fact, a reordering policy in the SSD controller gives reads a higher priority than writes, and even delays garbage collections and proceed reads to minimize potential interference [21]. However, this only guarantees the priority of reads at the channel level, thus not solving the read-blocking problem at the storage buffer, which occurs in the preceding layer of the channel. That is, the write-then-read serialization caused by read stalls in the upper buffer layer reduces the effect of the read priority at the controller.

The RAW protocol of S-Buf will also make multiple tenants interfere with each other in multi-tenancy. Even when multiple tenants issue their I/O requests independently, a read from one tenant will be stalled at the storage buffer by writes from other tenants. Thus, read-intensive tenants will be harmed by neighboring tenants with writes. In addition, as the in-storage write amplification increases over time, the detrimental effect of read stalls will exacerbate, resulting in worse application throughput and latency.

4.2 R-Buf

To address the challenges discussed in the previous section, we decouple the buffer resources for reads from writes so that reads are not interfered with by writes inside flash storage.

4.2.1 Key Idea. Selecting a dirty victim buffer for the replacement significantly increases overall read latency. This observation sparks an idea of choosing a clean victim instead of a dirty one. That is, we implement *CFLRU (Clean-First LRU)* [35] in S-Buf. CFLRU is a representative flash-aware buffer replacement algorithm that evicts clean pages first from the buffer to reduce the number of costly write operations. However, as will be detailed in Section 5.2.2, its effect is limited in solving the read stall problem of S-Buf.

This observation from CFLRU leads us to the idea of separating a data buffer for read requests. That is, the only way to ensure that read and write requests do not collide against the same resource is to make them use completely different resources. Thus, we propose a simple but effective buffer architecture, R-Buf. R-Buf is an in-storage DRAM buffer provisioned for read requests only, transferring read data between the host system and the flash NAND chips.

The design goals of R-Buf are i) to eliminate resource sharing between read and write requests, ii) to make reads faster by serving read requests in a read-dedicated buffer, and iii) to improve overall read latency from the storage buffer to the host application.

4.2.2 Read/Write Handling in R-Buf. Figure 5 overviews how I/O requests are handled in R-Buf. We omit the host interface controller and the flash controller in Figure 5 for simplicity. In R-Buf architecture, the in-storage buffer is split into two buffers: one for read requests (read buffer) and the other for write requests (write buffer). Both buffers are based on the LRU replacement policy. Let us detail how reads and writes are handled in R-Buf.

Read When the host application sends a read request for the page P_R, the host interface controller in SSD fetches the request and the FTL tries to retrieve P_R from the storage buffer. Both read buffer and write buffer manage the hash table by creating a hash entry for each buffer entry to speed up the search. Write buffer is searched first because it can contain the most recent version of P_R (❶). If there is no matching page in it, read buffer is searched (❷). If P_R is in either buffer, it is sent to the host system via a host DMA operation. Otherwise, P_R has to be read from the NAND chips. For that, the FTL first takes a buffer frame at the LRU tail in read buffer (❸). All buffer frames in read buffer are clean, so the victim data is

simply discarded. Then, the FTL generates a flash operation for a page read, then sends it to the flash controller to read P_R from the NAND flash array. The read data of P_R is stored in the secured data buffer using a flash DMA operation (❹). Finally, P_R is transferred to the host system through a host DMA operation (❺).

Write When the host sends a write request for the page P_W, the host interface controller fetches the request. In the same way as in handling reads, the FTL searches P_W from the storage buffer (❶ and ❷). If P_W is not in either buffer, it tries to get an available data buffer frame in write buffer for a page write. If the victim is clean, its data is discarded for reuse. This is a very rare case, right after the first write request comes in. Otherwise, the dirty data should be written to the NAND flash chips before being reused for P_W. For this, the FTL generates a flash operation for the dirty victim write, then sends it to the flash controller to write the victim data to the NAND flash chips (❸). In general, almost every buffer acquisition process will entail a dirty victim write because write buffer holds dirty data buffers generated by write requests. After buffer acquisition, P_W in the physical memory of the host system is transferred to the secured data buffer using a host DMA operation (❹). Finally, an interrupt is sent to the host system to indicate that the write for P_W is complete (❺). As the storage buffer is a write-back cache, P_W is transferred to the NAND flash array when it is evicted from the buffer.

To sum up, R-Buf resolves resource conflicts between reads and writes in the storage data buffer, so concurrent writes do not interfere with reads. Therefore, even in multi-tenancy, read requests from one tenant will not be blocked by writes from neighboring tenants. In addition, read-intensive tenants will be serviced at a constant time despite the increasing in-device write amplification.

Relation to Per-Channel Read Priority Because of the importance of read performance, flash controllers eagerly give priority to reads for queued I/O requests per channel [10, 31]. With S-Buf, however, such priority mechanisms can not realize their full potential. Often stalled at the storage buffer, read requests from the host will reach per-channel request queue slowly and thus have no chance to be prioritized. In contrast, with R-Buf, each read request from the host can be put to its corresponding per-channel queue at the earliest time, so it has chance to be prioritized over the already-queued writes. Fortunately, Cosmos+ OpenSSD supports the read priority mechanism [21], so reads from R-Buf can preempt the execution order over writes and even garbage collections that are already queued.

R-Buf with RW In handling an RW command, while securing buffer frames for P_W and P_R (steps ① and ③ in Figure 3, respectively), the victim storage buffer could be dirty. Therefore, the worst-case scenario is that both phases require the dirty victim to be emptied. To address this problem, we combine R-Buf and RW. This avoids the dirty victim eviction for P_R because read buffer in R-Buf only has clean data buffers. Furthermore, since there is no preceding write for P_R and the data buffers for reads and writes are separate, the read request via RW can be processed completely independently of the write request for P_W.

4.2.3 Prototype Implementation. We implement R-Buf in Cosmos+ OpenSSD [21]. Its DRAM space used as the shared buffer (S-Buf) is divided into two physically separate buffers: read and write buffer.

Given an I/O request, the FTL has first to perform a hash search for both buffers. Since both buffers work in write-back mode, there are four cases of buffer hit in R-Buf. For a read request, the FTL simply transfers the found data to the host, regardless of where the buffer hit occurs. For a write request, if it hits in write buffer, the FTL simply uses the found data. However, if it hits in read buffer, a slightly complicated process begins. Because two buffers are physically separated, moving a data buffer from one to the other is impossible. Therefore, we create a flag to indicate that the property of the data buffer temporarily changes. That is, we set the flag to temporarily switch the read data buffer frame to the write data buffer frame. Then, the FTL gives the highest priority to the flagged write request in the NAND scheduler. After the NAND write, the data buffer frame becomes a clean state, its flag is unset, and it is returned to read buffer.

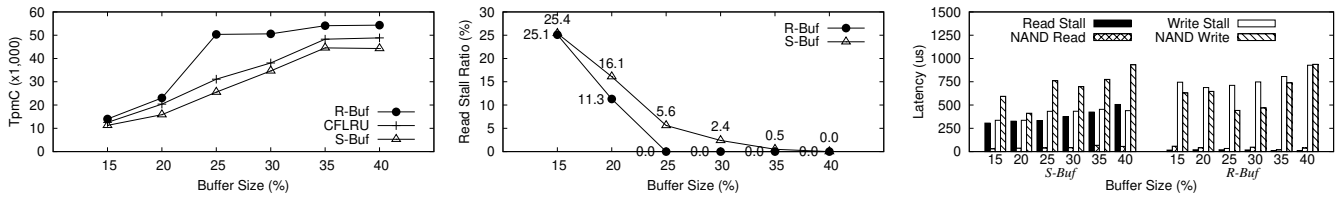
In very rare cases, two copies of the same page can reside in both buffers at the same time. In such a case, the FTL will return the latest copy in write buffer to guarantee consistency, and the old copy in read buffer can be safely discarded without causing any inconsistency.

Tuning the Size of R-Buf The Cosmos+ OpenSSD allocates 32MB of DRAM to S-Buf by default. In R-Buf, the size of read buffer might be critical to the I/O performance. To test the effect of read buffer size on I/O performance and also determine the optimal size empirically, we measure transaction throughput while running the TPC-C benchmark on MySQL with different read buffer sizes from 1MB to 16MB. We use the same TPC-C configuration used for Table 2a. The results indicates that throughput is almost the same regardless of read buffer size. This is because the TPC-C workload generates random reads to logical address spaces which is larger than the size of read buffer by several orders of magnitude. Hence, its read requests are unlikely to hit in read buffer. Instead, host-side DBMS buffer acts as an actual cache for read hits. Based on this empirical result, we decide to allocate 2MB to read buffer and the remaining 30MB to write buffer, respectively.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

We conduct all experiments on a Linux platform with 5.4 kernel. It is equipped with Intel Core i7-4770 CPU with 8 total cores and 32GB main memory. To evaluate the effect of RW and R-Buf on a real system, we implement them on Cosmos+ OpenSSD board [21]. The OpenSSD board includes two ARM Cortex-A9 cores on top of Xilinx Zynq-7000 board with 32GB MLC NAND flash memory. The board is connected to the host PC via PCIe interface, and the size of over-provisioning area is set to 10%. We use the `ext4` file system, and the direct I/O option (`O_DIRECT`) is enabled to minimize the interference from page caching of `ext4`. The benchmarking clients are run with database processes on the same hardware to avoid networking delays. Throughout all the experiments, unless otherwise stated, database size is 10GB, buffer cache size is 20% of the database size, the database page size is 4KB, and the number of concurrently running client threads is 8. To deliver steady-state



(a) Transaction Throughput **(b) Read Stalls at MySQL Buffer** **(c) I/O Latency at Storage Buffer**

Figure 6: TPC-C on MySQL: S-Buf vs. R-Buf

performance results, we always precondition the SSD using the dd command before the benchmark. Below are described the four workloads used in the experiments:

FIO FIO (Flexible I/O tester) is a synthetic tool that is used to generate various I/O patterns [3]. We use the random read and write workloads for benchmarking RW and R-Buf.

TPC-C tpcc-mysql from Percona [36] is used for TPC-C benchmark on MySQL. TPC-C is an industry standard OLTP workload for transactional database systems, consisting of heavy random reads and writes. For all TPC-C experiments, the initial database is set to 10GB (*i.e.*, 100 warehouses).

YCSB YCSB is a popular key-value store benchmark [7]. We use YCSB to benchmark R-Buf in MySQL on workloads with different read and update ratios.

DB-Bench DB-Bench is a standard benchmark framework to measure RocksDB performance [11]. We use the random I/O workload, `readwhilewriting`, performing random reads while writing.

TPC-H TPC-H is an OLAP benchmark consisting of 22 queries with various aggregation and join operations for large data set [41]. So, it is read-intensive. The benchmark is run on PostgreSQL to evaluate the effect of R-Buf on multi-tenant workloads.

5.1.1 Baseline RAW. Let us first clarify the baseline performance. DBMSs have many tunable options that control their run-time operation, and properly adjusting such knobs can achieve good performance [9, 37, 40, 42]. Hence, to make the background writer incessantly work, we tweak the knobs critical to I/O performance in MySQL [27, 29, 30]. For example, we tune the maximum IOPS used by the background writer to exploit high IOPS of SSDs. This tuned RAW improves throughput by 1.6x over the untuned RAW. It also reduces read stalls from 42% to 32%, but the ratio remains quite high due to the intrinsic structure of RAW. Given that knob tuning is an essential aspect of data-intensive applications [42], we use this knob-tuned RAW as a baseline throughout this paper.

5.2 Effect of R-Buf

5.2.1 FIO. To investigate the impact of read stalls on performance when writes co-run with reads, we run two *separate* FIO processes *simultaneously*: one issues only random reads of 4KB pages while the other does only random writes. Recall that, in Table 1, we execute reads and writes separately to measure each IOPS. As shown in Table 3, R-Buf has 5x higher read IOPS and 93% lower tail read latency than S-Buf. This clearly shows that writes can stall reads in S-Buf, worsening the read latency and, after all, that of the host application. On the other hand, R-Buf can eliminate read stalls at storage buffer, though sacrificing the write performance, achieving higher read IOPS and lower latency than S-Buf.

Table 3: FIO Performance: S-Buf vs. R-Buf

	S-Buf		R-Buf	
	Read	Write	Read	Write
IOPS (4KB)	5.4K	5349	28.1K	4156
Bandwidth (MB/s)	22.2	21.9	115	17.0
95th Latency (ms)	133	133	9	124

5.2.2 TPC-C. Next, in order to evaluate the effect of R-Buf on realistic database workloads, we measure various performance metrics while running the TPC-C benchmark on MySQL. We use different buffer sizes for MySQL, from 5% to 25% of the database size.

Transaction Throughput Before discussing the performance of R-Buf, let us look at the effect of CFLRU. In CFLRU, setting an appropriate value for LRU scan depth is important because it directly affects performance. For example, too large scan depth can increase the search overhead by scanning too deep to the target depth each time. Conversely, if the scan depth is too small, the value may not be sufficient to find a clean buffer, which may not properly validate CFLRU. To find a reasonable scan depth that can sufficiently capture the meaningful effect of CFLRU on Cosmos+ OpenSSD, we conduct a separate experiment with different scan depths and empirically set it to 64. As shown in Figure 6a, CFLRU shows 8-29% higher throughput than S-Buf, but its performance improvement is limited. This is because, the performance of CFLRU varies depending on the presence of a clean buffer within the scan depth. That is, CFLRU cannot avoid read stalls caused by shared buffer resources when no clean buffer is available in the scan depth.

On the other hand, R-Buf shows 21-97% better throughput than S-Buf and the performance gap remains as the buffer size increases to 40%. This clearly shows that removing read stalls in the storage buffer ultimately helps to improve the transaction throughput of the host application. Another observation from Figure 6a is that the performance of R-Buf spikes when the buffer pool size reaches 25% of the database size. The reason behind this performance spike is captured in Figure 6b. In R-Buf, the read stall at the DBMS buffer disappears when the buffer size reaches 25%. That is, after the buffer

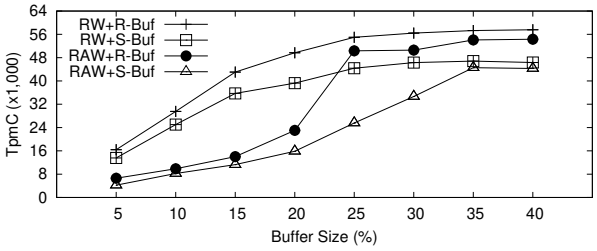


Figure 7: Effect of RW and R-Buf on TPC-C Throughput

size reaches 25% or more, read stalls in both storage buffer and host buffer are zero in R-Buf, resulting in a sharp increase in performance. In contrast, with S-Buf, read stalls disappear only when the buffer size reaches 40% of the database size. However, unlike R-Buf, it does not show remarkable performance improvement because even a larger host buffer cannot solve the storage-level read stalls.

Latency in Storage Buffer To validate the effect of R-Buf in terms of latency, we measure the time taken to get a free buffer frame at the storage buffer to service I/O requests and also the time taken to complete NAND I/Os, while running the same experiment used for Figure 6a, and present the result in Figure 6c. S-Buf has a much longer read stall latency than R-Buf, by at least 17x and up to 50x. This huge latency gap is because R-Buf can handle reads almost immediately, whereas S-Buf has to evict dirty victims for reads. With S-Buf, to be worse, the larger the MySQL buffer pool, the higher the read stall latency. With a larger buffer pool, the number of dirty pages in the buffer and the number of log writes increase due to active transaction processing, resulting in frequent checkpoint flushing. On the other hand, the number of read requests decreases because the buffer is large enough to cache the active data. As a result, the ratio of write requests to read requests increases, the number of dirty buffers in the storage buffer increases, and the probability of choosing a dirty buffer as a victim increases in S-Buf. Thus, the total read latency becomes longer in S-Buf.

Meanwhile, R-Buf has about 2x higher write latency than S-Buf. This is because the write buffer in R-Buf is full of dirty frames and thus, for every write request, a dirty victim has to be flushed. Though, since the transaction throughput is more dependent on the synchronous read performance, the longer write latency in R-Buf will be offset by the shorter read latency, as shown in Figure 6a.

5.3 Effect of RW Command

5.3.1 TPC-C Performance.

Throughput In R-Buf, when the buffer size is same to or larger than 25% of the database size, read stalls at database buffer disappear. Therefore, in order to evaluate the performance of RW, we conduct the same TPC-C experiment using relatively small buffer that causes read stalls in the database buffer. That is, we perform the experiment by changing the buffer size of MySQL from 5% to 25% of the database size. In addition, to compare the performance when read stalls are removed from the host and/or storage buffers, we test all combinations of RW and R-Buf. The results are presented in Figure 7.

Table 4: I/O Throughput: Vanilla vs. RW+R-Buf

Performance Metrics	RAW+S-Buf	RW+R-Buf
TpmC	15,857	49,668
Read IOPS	2,778	7,174
Write IOPS	2,477	4,901

We first analyze the effect of RW on S-Buf. In S-Buf, RW (RW+S-Buf) can sustain 1.7x to 3.2x higher throughput than RAW (RAW+S-Buf). In particular, as presented in Table 2b, the smaller the MySQL buffer size, the more frequent read stalls occur. RW can avoid these read stalls, so the relative performance gap between RW and RAW becomes more significant with the smaller buffer size. In addition, in RW, the reduction in costly tasks such as interrupts and context switches contributes to the higher throughput of RW. On the other hand, when the buffer size becomes larger than 35%, the performance gap between RW and RAW shrinks since read stalls rarely occur at large database buffer, as presented in Figure 6b.

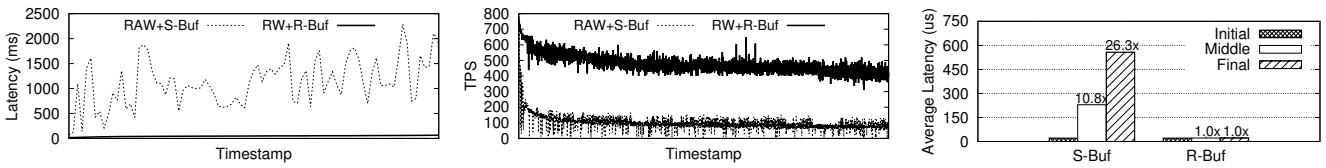
Next, we discuss the effect of RW in R-Buf. With the small MySQL buffer size (*e.g.*, less than 20% of database size), R-Buf alone (RAW+R-Buf) achieves only a minor performance gain over S-Buf (RAW+S-Buf). The main reason is that a large fraction of reads are stalled at the upper MySQL buffer layer and thus only a small number of reads can benefit from R-Buf. In contrast, when RW are used together with R-Buf (RW+R-Buf), they eliminate read stalls at both DBMS buffer and storage buffer, improving the throughput up to 3.9x over RAW (RAW+S-Buf).

I/O Throughput To validate that RW and R-Buf can utilize the flash storage better than RAW and S-Buf, we measure read and write IOPSs using `iostat` utility in the Linux while running the TPC-C benchmark in Figure 7 in two modes, (RAW+S-Buf) and (RW+R-Buf). Table 4 summarizes the average read and write IOPSs. For a comparison, the TpmC metric is also presented in the table.

As shown in Table 4, all measured IOPS values are consistent with transaction throughput (*i.e.*, TpmC). Specifically, (RW+R-Buf) shows 3.1x better transaction throughput, 2.6x higher read IOPS, and 2.0x higher write IOPS than (RAW+S-Buf). These results confirm that RW and R-Buf make flash storage better utilized, ultimately increasing throughput of the host application.

System Metrics To confirm the benefit of RW in reducing costly system tasks at run-time, we collect performance counter statistics, including interrupts, context switches, and CPU instructions, while running the TPC-C benchmark in two modes: (RAW+S-Buf) and (RW+R-Buf). For this, we use the Linux tools, `perf` and `vmstat`. In the experiments, the buffer size is set to 20% of database size. We calculate the number of each system metric per transaction and present them in Table 5.

As shown in Table 5, (RW+R-Buf) reduces the number of interrupts by 41% and context switches by 51% compared to (RAW+S-Buf). Since RW issues read and write as a single command and returns success only once to the host, the number of interrupts sent to the host is reduced, and accordingly, that of context switches decreases. This can save CPU cycles and memory requirements for them; thus, the host application can process more transactions



(a) 99th Transaction Latency **(b) Transaction Throughput** **(c) In-Storage Read Latency**

Figure 8: WAF and Sustained Performance (TPC-C, Buffer Size = 20%): Vanilla vs. RW+R-Buf

Table 5: System Metrics (TPC-C, Buffer Size = 20%)

Metrics (per Transaction)	RAW+S-Buf	RW+R-Buf
Interrupts	12.12	7.21
Context Switches	30.03	14.73
CPU Instructions	1,784K	1,227K

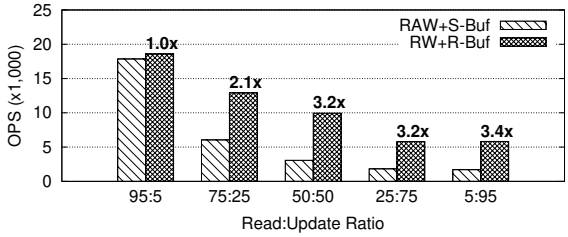


Figure 9: Varying Read:Update Ratio: Vanilla vs. RW+R-Buf

faster. It should be noted that considerable reduction in such metrics by (RW+R-Buf) purely explains the throughput gap between (RW+R-Buf) and (RAW+R-Buf) in Figure 7. Meanwhile, the throughput gap between (RAW+R-Buf) and (RAW+S-Buf) in the figure is attributable to the read stall at the flash storage buffer.

Another observation from Table 5 is that (RW+R-Buf) can considerably lower the number of CPU instructions to 31% compared to (RAW+S-Buf). The main reason is that reduced I/O calls decrease the CPU instructions used by the I/O stack. RW also avoids unnecessary CPU works required for the *write-then-read* process, such as acquiring a free buffer and issuing write and read requests, respectively. For example, RW can contribute to reduce the number of mutex operations, as mentioned in Section 3.2.2. To quantify the mutex overhead, we measure the average number of mutex operations for the free list per transaction, and the results are 190 for RAW and 114 for RW. Converting to wait time for the mutex, it takes 25us in RAW while it does 13us in RW. That is, RW can save the per-transaction mutex wait time by 12us. Consequently, the simplified buffer manager of RW can complete one transaction with quite less CPU instructions.

Varying Read and Write Ratio Recalling that the goal of both RW and R-Buf is to alleviate read stalls caused by the slow writes, its effect is highly dependent on the relative ratio of read and write operations issued from workloads. In particular, its effect will reduce as the fraction of reads over writes increases because the larger read ratio results in fewer pending writes and fewer stalled reads. To demonstrate this, we conduct the YCSB benchmark by changing the ratio of read and update operations from 5% to 95%, respectively. The results are presented in Figure 9.

(RW+R-Buf) shows better throughput than (RAW+S-Buf) except for the extremely read-intensive case (*i.e.*, `Read:Update=95:5`). As expected, the relative performance gain of (RW+R-Buf) over (RAW+S-Buf) shrinks as the read ratio increases and thus read stalls decrease. On the other hand, the effect of (RW+R-Buf) stands out as the update ratio increases. In write-intensive workloads (*e.g.*, `Read:Update=5:95`), dirty victim evictions from both host and storage buffers also increase on (RAW+S-Buf), so read performance ultimately determines the overall performance. Therefore, (RW+R-Buf) can reduce the 99th percentile latency by 87% for read operations and 74% for update operations, compared to (RAW+S-Buf). This results leads to 3.4x better throughput in (RW+R-Buf).

5.3.2 Other Workloads: LinkBench and SysBench. To verify the effect of (RW+R-Buf) on other workloads, we run SysBench [20] and LinkBench [2] with the same configuration as Figure 7. Though not presented pictorially because of the space limit, the results show that the relative performance gap between (RW+R-Buf) and (RAW+S-Buf) is almost the same as that of TPC-C. This is rather obvious recalling that the read-to-write ratios in three workloads are almost the same.

5.4 WAF and Sustained Performance

The random write performance in flash storage is highly dependent on the size of over-provisioning area available for garbage collection [22]. To understand the I/O interference impact over time, we measure WAF, 99th transaction latency, and transaction throughput while running the TPC-C benchmark on MySQL for 57 hours until the initial database of 15GB fills the remaining 17GB space.

5.4.1 WAF and DB Performance. It is well known that with flash storage, as the database grows and fills the capacity, the over-provisioning area in effect decreases, resulting in high WAF and penalizing writes [17]. Therefore, the read performance of (RAW+S-Buf) is severely degraded on read stalls due to the preceding long-write-latency, so its tail transaction latency increases over time, as shown in Figure 8a. In addition, the latency fluctuates significantly because the read latency is highly dependent on the state of the victim buffer (*i.e.*, clean or dirty). Consequently, unstable read latency drops TPS from 500 to 80, as shown in Figure 8b. This result is

consistent with that on commercial SSDs [17]. In contrast, (RW+R-Buf) has a constant transaction latency of 90% lower than vanilla on average. This clearly indicates that (RW+R-Buf) does not suffer from read stalls, resulting in 4.9x higher throughput. Furthermore, note that, at the end of the benchmark, the throughput gap between the two (*i.e.*, 5.3x) is larger than that in Figure 7 (*i.e.*, 3.1x) due to the impact of the write penalty caused by the increased WAF.

5.4.2 WAF and Read Latency in Storage Buffer. To quantify the amount of time spent inside storage buffer when processing reads, we measure the average in-storage read latency at three time points during the TPC-C benchmark. In detail, we insert a time-measuring function for OpenSSD firmware (`XTime_GetTime()` of Xilinx) before and after every read request execution to measure the elapsed time from fetching a host read request to completing it.

As shown in Figure 8c, at the initial point of the benchmark, the average in-storage read latency in S-Buf is only 21us because of very few dirty victims in it. However, over time, the number of dirty victims increases due to write requests from the TPC-C, so the probability of selecting them for eviction also increases—54% of read requests entail dirty victim writes. Besides, ever-increasing WAF makes the penalty of read stalls high. As a result, at the final point of the benchmark, the latency is increased by 26x (*i.e.*, 557us). This long in-storage read latency, in turn, increases the transaction latency at the host application. Meanwhile, R-Buf shows a constant read latency of around 21us. Read requests are handled immediately using clean data buffers in read buffer, so they are not affected by ever-increasing WAF. Consequently, R-Buf can process read requests at the native speed of reads, guaranteeing stable read latency at both in-storage and the host application layers.

5.5 Other Cases for R-Buf

Till now, we have shown how much traditional relational DBMSs can benefit from RW and R-Buf. Meanwhile, since R-Buf enables reads not to be interfered with by concurrent writes, it can benefit other cases where reads and writes are concurrently issued from multiple and independent processes. This section demonstrates that R-Buf alone is useful for LSM-based KV-stores and multi-tenancy. With the ever-growing capacity of SSDs, it is not uncommon for multiple databases to share a single large SSD, which is particularly true for the cloud environment [8, 25]. Under multi-tenant workloads, S-Buf will stall read requests from one tenant due to concurrent write requests from neighboring tenants. In contrast, R-Buf can isolate noisy neighboring tenants in resource sharing because it can eliminate the resource conflict between concurrently issued read and write requests inside the same storage. For the multi-tenancy experiments, we create two disk partitions and run two tenants concurrently on each partition.

RocksDB To check how R-Buf responds to mingled reads and writes in a popular LSM-based KV-store, RocskDB, and affect the latency and throughput, we measure the 99th read latency and throughput while performing random I/O using `readwhilewriting` of DB-Bench. Figure 10 shows the results.

We first notice that the 99th read latency of R-Buf is 41% lower than that of S-Buf. Furthermore, R-Buf achieves 1.7x better throughput compared to S-Buf. In RocksDB, as more and more data is written and updated over time, multiple versions for the same key can

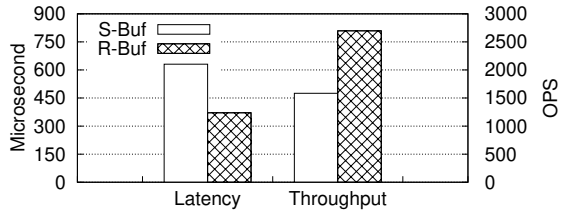


Figure 10: DB-Bench on RocksDB: S-Buf vs. R-Buf

exist across multiple levels (*i.e.*, SST files). To remove these copies, RocksDB periodically deletes duplicate keys and merges them, and this is called compaction. To carry out compaction, RocksDB has by default four background processes with sequential read and write access patterns. Meanwhile, multiple foreground processes will issue random reads to the storage to fetch data. And the latency of such reads will determine the transaction latency as well as throughput. With R-Buf, random reads from foreground processes are not stalled by the concurrent compaction writes. However, such reads can be stalled with S-Buf. For this reason, R-Buf outperforms S-Buf in the experiment.

TPC-C and TPC-H Next, we demonstrate the performance improvement of R-Buf in multi-tenancy. First, we concurrently run two tenants—random I/O-intensive TPC-C (on MySQL) and read-only TPC-H (on PostgreSQL)— and observe the impact of the increased WAF by TPC-C on read-only queries in TPC-H.

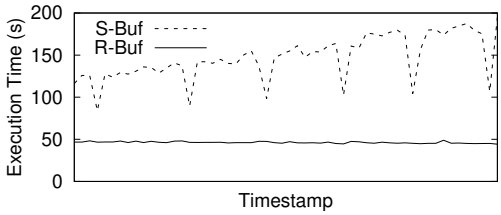


Figure 11: Query Time of Q1@TPC-H (Co-run with TPC-C)

Figure 11 shows the execution time of Query 1 in TPC-H over 11 hours. First, the query time of R-Buf is steady, whereas that of S-Buf constantly increases over time by up to 4x. In a separate experiment using the commercial SSD, `SSD-D`, the query execution time behaves almost same as that in S-Buf in Figure 11. At the beginning of the benchmark, R-Buf already shows a shorter query execution time than S-Buf. This is because R-Buf is, unlike S-Buf, free from the read stall problem. Furthermore, as the TPC-C database continues to grow over time, the increased WAF prolongs the write latency and thus exacerbates the read stall problem in S-Buf, increasing the query time of TPC-H. In detail, the total query time of TPC-H in S-Buf increases from 309s to 381s while remaining almost constant at 222s in R-Buf. Meanwhile, R-Buf drops the throughput of TPC-C just by 7%. Due to the write buffer in R-Buf, R-Buf suffers a little performance loss in TPC-C, but in TPC-H, it greatly increases the query performance compared to S-Buf. Consequently, R-Buf can make two tenants perform more *proportionally* than S-Buf and better utilize the parallelism of flash storage.

Table 6: Multi-Tenancy Performance: TPC-C and DB-Bench

Performance Metrics	S-Buf	R-Buf
TpmC (TPC-C)	4,184	3,624
OPS (DB-Bench)	281	704
In-Storage Read Latency (us)	293.4	21.7

TPC-C and DB-Bench Finally, we study the multi-tenancy performance of two I/O-intensive tenants using different database engines: TPC-C on MySQL and DB-Bench on RocksDB. Table 6 shows the throughput of each tenant and the in-storage read latency.

In TPC-C, the transaction throughput of R-Buf decreases by 13% compared to S-Buf. In R-Buf, every write request from MySQL entails a dirty victim write and thus takes a long time to complete, while read requests proceed at no stall. Furthermore, since RocksDB issues a lot of batch reads and writes, TPC-C will experience more I/O interference than when running as a single tenant. On the other hand, the throughput of R-Buf in DB-Bench is 2.5x higher than that of S-Buf. Especially for RocksDB, R-Buf has 90% lower 99th percentile read latency and 2% higher write tail latency than S-Buf. This is because the sequential writes of RocksDB are less affected by random I/O of TPC-C [5]; thus, the write performance loss of DB-Bench is less than TPC-C. In addition, the interference between reads and writes increases the in-storage read latency of S-Buf by 13.5x compared to R-Buf. Note that R-Buf has the same in-storage read latency as single tenancy.

6 RELATED WORKS

In that RW and R-Buf are flash-aware solutions for host buffer and storage buffer, respectively, four types of existing works are related: flash-aware buffer management at the host layer [32, 35] and at the SSD layer [19, 44], read prioritization in flash storage controller [21, 31], and flash-aware database systems [15, 23, 34]. Neither one, however, addresses the read stall problem. RW and R-Buf are unique in that they focus on serving reads faster by resolving RAW-induced read stalls at both buffers.

Flash-aware Buffer Management at the Host Several flash-aware buffer management schemes have been suggested to address the read and write asymmetry [32, 35]. For instance, Clean-First LRU (CFLRU) [35] divides the host buffer space into working region and clean-first region and preferentially selects victim pages for replacement from the clean-first region. Trading fast reads to reduce the number of slow writes and thus compromising the hit ratio, it aims at optimizing the total cost of accessing flash storage. However, unlike RW and R-Buf, CFLRU is not designed to remove read stalls. It simply assumes the RAW protocol upon dirty victims.

Buffer Management in SSDs A few recent research have investigated buffer management inside SSDs. CBM [44] focuses on write buffer inside SSDs to reduce the negative impact of random writes. It uses DRAM as a read cache to speed up read accesses and NVM to manage dirty pages. R-Buf and CBM have in common that they split the shared data buffer in two, but CBM focuses on using the write buffer more effectively. BPLRU [19] allocates the DRAM buffer inside the SSD only for write requests and simply redirects read requests to the FTL. Since BPLRU is primarily concerned with random write performance, it does not consider read requests and excluded reads from the experiments. Unlike BPLRU, we mainly focus on read requests; thus, it is hard to compare them on equal terms. In summary, both CBM and BPLRU pay attention to the performance of random writes, and they do not address the read stall problem in the storage buffer layer.

Prioritizing Read in Flash Controller The read operation has been prioritized over the write operation at the flash storage controllers [10, 31] because it is critical to the latency as well as the throughput of I/O-intensive applications. To be specific, to reduce the read latency caused by the read-write interference, the preceding write or even erase operations enqueued at a flash chip can be suspended in favor of serving the following reads. Though effective in reducing read latency [46], however, the read prioritization technique is orthogonal to the read stall problem in the storage buffer. For example, although the Cosmos+ OpenSSD support read prioritization [21], the read stall problem still persists on it, as shown in Figure 4 (`S-Buf`). It should be recalled that the prioritization technique is intended to resolve the interference between writes and reads at the channel level of the SSD but not to address the read stall problem.

Flash-aware Database Systems A few proposals about flash-aware database systems have been recently made [15, 23, 34]. One common design objective among them is to minimize or even remove the I/O stack overhead on flash storage with short I/O latency. However, they do not address the read stall problem at all. Considering that LeanStore [23] and DANA [15] are based on the background writer and latency-critical reads, we assume that the read stall problem still exists in such flash-aware DB systems. Hence, they will benefit from RW and R-Buf. In particular, in the case of SaS [34] which should also undergo read stalls in its current form, it can embody the idea of RW command by directly leveraging the DRAM buffer as the resource to resolve read stalls.

7 CONCLUSION

This paper confirms that the RAW protocol and its resulting read stall are sub-optimal on flash storage. To solve the problems, we propose two simple but effective solutions: RW and R-Buf. They allow performance-critical reads to proceed without being blocked by slow writes in the DBMS buffer and storage buffer, respectively, complementing each other. Thus, with the help of RW, R-Buf, and the read priority mechanism at the flash channel, reads do not experience any interference from writes on the I/O path from the host buffer manager to the flash chips. As a result, they improve transaction throughput, latency, and storage utilization.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00314, NVRam Based High Performance Open Source DBMS Development) and by Samsung Electronics. We are thankful to the anonymous reviewers for their insightful feedback.

REFERENCES

[1] Ibrar Ahmed, Gregory Smith, and Enrico Pirozzi. 2018. *PostgreSQL 10 High Performance: Expert Techniques for Query Optimization, High Availability, and Efficient Database Maintenance.* Packt Publishing.

[2] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* 1185–1196.

[3] Jens Axboe. [n.d.]. FIO (Flexible IO Tester). https://github.com/axboe/fio.

[4] William Bridge, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza, and N. MacNaughton. 1997. The Oracle Universal Server Buffer. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 590–594.

[5] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)* 12 (2016), 1 – 39.

[6] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. 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.* 266–277. https://doi.org/10.1109/HPCA.2011.5749735

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10).* 143–154.

[8] Intel Corporation. 2018. Accelerated SSD Infrastructure for the Cloud. https://builders.intel.com/docs/datacenterbuilders/accelerated-ssd-infrastructure-for-the-cloud-with-attala.pdf. (2018).

[9] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR.*

[10] Nima Elyasi, Changho Choi, Anand Sivasubramaniam, Jingpei Yang, and Vijay Balakrishnan. 2019. Trimming the Tail for Deterministic Read Performance in SSDs. In *2019 IEEE International Symposium on Workload Characterization (IISWC).* 49–58. https://doi.org/10.1109/IISWC47752.2019.9042073

[11] Facebook. 2014. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[12] Jim Gray and Bob Fitzgerald. 2008. Flash Disk Opportunity for Server Applications: Future Flash-Based Disks Could Provide Breakthroughs in IOPS, Power, Reliability, and Volumetric Capacity When Compared with Conventional Disks. *Queue* 6, 4 (July 2008), 18–23. https://doi.org/10.1145/1413254.1413261

[13] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Guy Harrison. 2014. Using Flash SSD to Optimize Oralce Database Performance. https://www.slideshare.net/gharriso/ssd-and-the-db-flash-cache.

[15] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020.*

[16] Jasmine OpenSSD. 2011. OpenSSD Project. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[17] Minji Kang, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2020. 2R: Efficiently Isolating Cold Pages in Flash Storages. *Proceedings of VLDB Endowment* 13, 12 (jul 2020), 2004–2017.

[18] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2016. Flash as Cache Extension for Online Transactional Workloads. *The VLDB Journal* 25, 5 (Oct. 2016), 673–694. https://doi.org/10.1007/s00778-015-0414-1

[19] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (San Jose, California) *(FAST'08).* USENIX Association, USA, Article 16, 14 pages.

[20] Alexey Kopytov. 2018. SysBench. https://github.com/akopytov/sysbench.

[21] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Transactions on Storage* 16, 3, Article 15 (July 2020).

[22] Sang-Won Lee, Bongki Moon, and Chanik Park. 2009. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09).* 863–870.

[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* 185–196.

[24] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93).* 22–31. https://doi.org/10.1145/170035.170042

[25] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *PVLDB* 12, 12 (2019), 1942–1945.

[26] Violin Memory. 2016. Flash Fabric Architecture (Version 2.0). A Whitepaper from Violin Memory.

[27] MySQL Team (Oracle Corp.). 2021. Configuring Buffer Pool Flushing. https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-flushing.html.

[28] MySQL Team (Oracle Corp.). 2021. The InnoDB Buffer Pool. https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html.

[29] MySQL Team (Oracle Corp.). 2021. Optimizing InnoDB Disk I/O. https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-diskio.html.

[30] MySQL Team (Oracle Corp.). 2021. Server System Variable Reference. https://dev.mysql.com/doc/refman/5.7/en/server-system-variable-reference.html.

[31] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. 2011. Ozone (O3): An Out-of-Order Flash Memory Controller Architecture. *IEEE Trans. Comput.* 60, 5 (2011), 653–666. https://doi.org/10.1109/TC.2010.209

[32] Sai Tung On, Shen Gao, Bingsheng He, Ming Wu, Qiong Luo, and Jianliang Xu. 2014. FD-Buffer: A Cost-Based Adaptive Buffer Replacement Algorithm for FlashMemory Devices. *IEEE Trans. Comput.* 63, 9 (2014), 2288–2301. https://doi.org/10.1109/TC.2013.52

[33] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)* (Virtual Event, China) *(DAMON'21).* Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/3465998.3466003

[34] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2021. SaS: SSD as SQL Database System. *Proceedings of VLDB Endowment* 14, 9 (may 2021), 1481–1488.

[35] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea) *(CASES '06).* Association for Computing Machinery, New York, NY, USA, 234–241. https://doi.org/10.1145/1176760.1176789

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

[37] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-Tuning Memory in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06).* VLDB Endowment, 1081–1092.

[38] Steven Swanson and Adrian Caulfield. 2013. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer* 46, 8 (Aug. 2013), 52–59. https://doi.org/10.1109/MC.2013.222

[39] J. Z. Teng and R. A. Gumaer. 1984. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal* 23, 2 (1984), 211–218. https://doi.org/10.1147/sj.232.0211

[40] The PostgreSQL Global Development Group. 2019. PostgreSQL 11 Documentation: Resource Consumption. https://www.postgresql.org/docs/current/runtime-config-resource.html.

[41] TPC. [n.d.]. TPC-H. http://www.tpc.org/tpch.

[42] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* 1009–1024.

[43] Daniel Waddington and Jim Harris. 2018. Software Challenges for the Changing Storage Landscape. *Commun. ACM* 61, 11 (oct 2018), 136–145. https://doi.org/10.1145/3186331

[44] Qingsong Wei, Cheng Chen, and Jun Yang. 2014. CBM: A cooperative buffer management for SSD. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST).* 1–12. https://doi.org/10.1109/MSST.2014.6855545

[45] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. When Storage Response Time Catches Up With Overall Context Switch Overhead, What Is Next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4266–4277. https://doi.org/10.1109/TCAD.2020.3012322

[46] Guanying Wu and Xubin He. 2012. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12).* USENIX Association, USA, 10.