WAFLASH: Taming Unaligned Writes in Solid-State Disks

Shuibing He[♮] Matthew Myers[†] Xuehao Duan[*] Keegan Sanchez[†] Xuechen Zhang[†]

[♮]College of Computer Science and Technology
Zhejiang University

[†]School of Engineering and Computer Science
Washington State University Vancouver

[*]School of Computer Science
Wuhan University

2022 IEEE International Conference on Networking, Architecture and Storage (NAS) | 978-1-6654-5408-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/NAS55553.2022.9925375

***Abstract*—NAND-flash based solid-state disks (SSDs) are replacing the hard-disk drives (HDDs) in various storage systems from high-end servers in data centers to mobile computers on the edges of cloud computing. Due to the architectural nature of SSDs, performance-sensitive applications may generate a large number of unaligned writes on SSDs, which can cause many issues including chip congestion, sub-request blocking, chip load imbalance, and write space amplification. This paper presents a Write-Aligned FLASH drive (WAFLASH) that comprehensively and significantly alleviates the side-effects of unaligned writes in solid-state disks. We utilize three key techniques: (1) prioritizing eviction of fully-filled pages over partially-filled pages in write buffers, (2) storing multi-version data requested by unaligned writes/overwrites in flash memory, and (3) compacting multiple small partially-filled pages in a physical page to circumvent write amplification and reduce the number of additional reads in the critical I/O path. We implement WAFLASH in VSSIM and Cosmos+ OpenSSD. The results show WAFLASH increases I/O throughput by up to 125% with the Filebench benchmark.**

***Index Terms*—Flash, Page Unalignment, FTL, Buffer Management**

I. INTRODUCTION

NAND flash-memory-based SSDs are adopted as an increasingly important storage medium for supporting applications that demand low I/O latency or high I/O throughput [1]. They are increasingly used as hard disks (HDDs) replacement for both scientific and enterprise applications. The SSD market is continuously growing at a significant rate [2], and flash/SSD-based storage becomes a norm in various storage systems from high-end servers in data centers [3], [4] to mobile computers on the edges of the cloud computing [5].

Although other interfaces exist, most of the existing systems use the legacy block interface to access SSDs for portability and compatibility with HDDs. The size of I/O requests issued from OSs are generally aligned with the page size (e.g., 4 KB) of virtual memory. It can be much smaller than the flash page size (e.g., 8 KB or 16 KB) in SSDs. With this architectural nature, applications may generate a large number of writes that are not aligned with the flash page boundary of SSDs. We call them *unaligned writes* in this paper. As shown in § II, the unaligned writes are common with practical I/O workloads.

With the increase in flash density, the flash page size can also increase to 32 KB or even larger [6]. This makes the unaligned problem more severe because existing systems access SSDs without knowing the change in page sizes. Furthermore, when direct I/Os are used, requests to SSDs bypass OS buffers and arrive at SSDs at the unit of a sector. As a result, they may be not aligned with the page size of virtual memory. Nor are they aligned with the flash page size. Therefore, using direct I/Os may further increase the ratio of unaligned writes.

Serving unaligned writes in SSDs can significantly degrade system efficiency for several reasons. First, it may cause chip congestion. SSDs need to write partial pages[1] for serving the unaligned writes. Writing partial pages may incur read-modify-write operations, increasing the number of requests in chips. Second, it may lead to chip waiting because the additional reads need to wait for previous requests on the same chip [7]. The waiting time can be much longer than the actual service time of the reads. Third, it may result in a load imbalance across the chips [8]. To achieve high parallelism, a large request in SSDs is decomposed into a number of sub-requests over multiple chips. If one of the sub-requests involves writing a partial page, the other sub-requests must wait for its completion because a request is not complete until all its sub-requests are complete. Fourth, it may cause low page utilization because of the write space amplification.

Several existing approaches have been proposed to address the side-effects of unaligned writes. MCA (Multi-Chip based replacement algorithm) is the state-of-the-art scheme that focuses on reducing chip waiting with new buffer management [7]. Slacker can alleviate load imbalance across chips by considering the completion time of each sub-request of a large request in I/O scheduling [8]. Lu et al. use an object interface to replace the block interface for SSD accesses [9]. However, neither the buffer management nor I/O-scheduling based approaches can comprehensively resolve all the aforementioned side-effects of unaligned writes. Adopting a new object interface requires a significant amount of programming effort, which may not be possible for legacy applications.

In this work, we design a new flash drive WAFLASH that significantly alleviates the side-effects of unaligned writes to flash memory with cooperative management of SSD write

[1]We use *partial pages* to refer partially-filled flash pages where some sectors in the page are not updated and *full pages* to refer flash pages that all sectors in the pages are to be updated.

978-1-6654-5408-7/22/$31.00 ©2022 IEEE

buffer and its flash translation layer (FTL). Specifically, we propose a Partial page and Congestion-aware LRU (PC-LRU) buffer management algorithm, which differentiates writes to partial pages from those to full pages. Without compromising data locality, PC-LRU prioritizes the eviction of full pages over partial pages in the buffer when chip congestion happens to reduce the number of unaligned writes sent to FTL. Furthermore, to circumvent read-modify-write operations for serving unaligned writes, we design a new multi-version FTL, called MV-FTL. It serves a partial-page write just as a write to a full page by directly writing the data to a new flash page without performing the read and modification operation. Instead of invalidating the old version of the data, MV-FTL keeps both of the copies valid until they are merged. We use MV-FTL as an index structure to access both versions of data in flash. Due to the out-of-place update nature of flash, MV-FTL does not need additional space for keeping the two versions of the data. Finally, when unaligned writes in MV-FTL cause low flash page utilization, we propose a novel data compaction technique to merge multiple partial pages, therefore, increasing page space utilization and reducing the number of unaligned writes to flash memory.

Our contributions are summarized as follows:

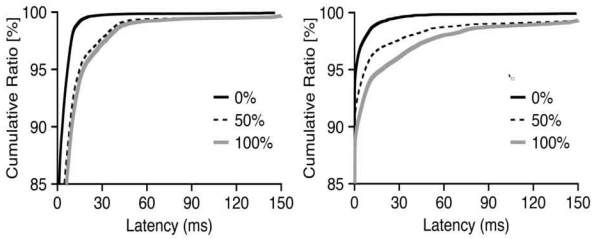
- We propose a novel buffer replacement algorithm, PC-LRU, that prioritizes the destaging of full pages over partial pages when chip congestion happens and transforming partial pages to full pages to reduce the number of unaligned writes to flash.
- We design a new flash translation layer, MV-FTL, that supports multi-version data management upon serving unaligned writes/overwrites when the write buffer is full without incurring extra reads.
- We propose a partial page compaction method to optimize data storage in MV-FTL for reducing the space overhead and the number of unaligned writes.
- We implement WAFLASH using the VSSIM [10] platform based on QEMU/KVM to run real file systems and applications. The results show that with our method the average latency and I/O throughput are improved by up to 62% and 125%, respectively.

II. Background and Motivation

A. SSD and Flash Translation Layer

SSDs use flash memory as a storage medium. Flash memory consists of blocks, each further consisting of 64 to 256 pages. Each page has a data area (1 to 16 KB) and a meta-data area (called out-of-band (OOB) data) [11]. Unlike traditional disks, flash memory reads and writes data in the unit of a page and erases in the unit of a block. Compared to read latency, flash has a much longer write and erase latency. In addition, blocks must be erased before they can be reused, thus SSDs usually perform out-of-place updates.

SSDs use the FTL to support hosts to access flash memory via the block interface as conventional HDDs. Hosts send requests to SSDs in the unit of a sector (e.g., 512 B or 4 KB). Request sizes are mostly aligned with page size of virtual memory of OSs. The commonly used page size of OSs is 4 KB. Therefore, the request size can be much smaller than a flash page size (e.g., 8 KB and 16 KB). The FTL performs address mapping, garbage collection, and wear leveling. Furthermore, SSD controller often uses RAM as a buffer on top of the FTL to speed up write performance because flash writes are relatively slow compared to reads. In this paper, we aim to significantly reduce the negative impact of unaligned writes to flash by more efficiently utilizing the buffer and the FTL.



(a) read (b) write

Fig. 1: The impact of serving unaligned writes on SSD performance. Both read and write latency become longer.

B. Motivation

By analyzing I/O traces from various computing environments we have confirmed that unaligned writes are common for data-intensive applications when accessing SSDs. Specifically, we study the ratio of unaligned writes in the set of block-level traces from system software (e.g., MSNFS, CFS) [12] and mobile applications (e.g., Nexus5) [12], [13] when the flash page size is 8 KB and 16 KB. For traces collected before 2010, we assume its sector size is 512 B. Otherwise, we assume that its sector size is 4 KB.

The average ratio of unaligned writes is 83.1% with an 8 KB flash page size. Even worse, the trend of increasing flash page size can further increase the ratio of unaligned writes. For example, when the flash page size is increased from 8 KB to 16 KB, the ratio of unaligned writes in the MSNFS trace will increase from 33.5% to 99.1% and the average ratio for all workloads reaches 97.2%.

To experimentally investigate the effects of unaligned writes on flash read/write performance, we conduct experiments on a simulated SSD [14] by feeding it with 120K requests from the MSNFS workload. The flash page size is 8 KB and an optimum page-level FTL is deployed. 30% of the requests are reads and the rest are writes. All the read requests are 8 KB and aligned with page boundaries. But for writes, we make a portion of them unaligned to flash page boundaries. We vary the ratio of the number of unaligned writes to the number of total writes from 0% to 100%.

Figure 1a and 1b plot the latency CDF of all reads and all writes, respectively. One can observe that more unaligned write noises lead to longer tail latencies for both reads and writes. For example, with 50% noise write latencies are $8x$, $24x$ longer compared to no write noise cases, at 90^{th} and 95^{th}

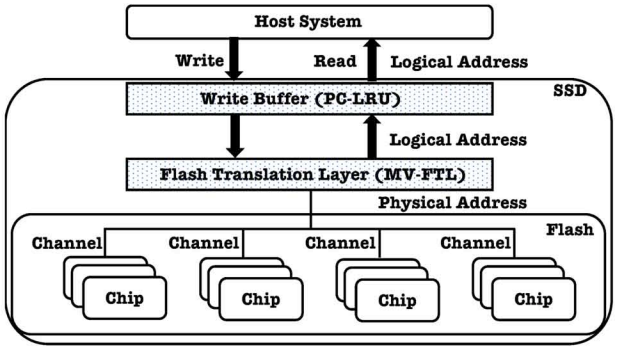


Fig. 2: System architecture. The proposed write buffer management scheme (PC-LRU) and multi-version-aware FTL (MV-FTL) are applied to the write buffer and FTL layer inside SSDs respectively.

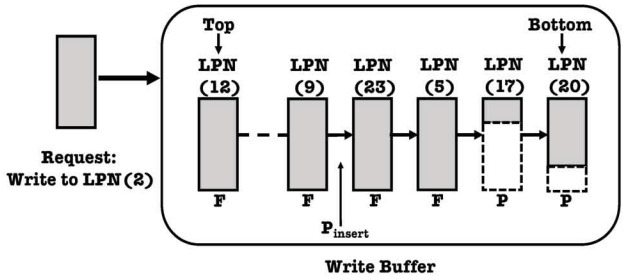


Fig. 3: PC-LRU write buffer management. LPN: logical page number. We assume 1 flash page consists of 4 sectors. $LPN(2)$, $LPN(12)$, $LPN(9)$, $LPN(23)$, and $LPN(5)$ are full-page writes (marked as F) and $LPN(17)$ and $LPN(20)$ are partial-page writes (marked as P). When chips are idle, upon serving the write to the full page $LPN(2)$, the partial pages $LPN(17)$ and $LPN(20)$ at the bottom of the LRU stack are replaced; when chips are not idle, the full page $LPN(5)$ is replaced.

percentiles, respectively. Therefore, it is necessary to eliminate their side-effects to improve SSD performance.

III. The Design of WAFlash

We present the design of WAFlash, a new SSD architecture that can comprehensively alleviate the side-effects of unaligned writes. Figure 2 shows the general system architecture considered in this paper. In the paper, we assume the host system issues I/O requests to SSDs at the unit of a sector. The RAM buffer in SSDs is used only for write requests because flash reads are much faster than writes and repeated reads can be effectively absorbed by the host cache [15]. The buffer is allocated to write requests at the unit of a sector to save RAM space. The read requests missed in the buffer are directed to the flash translation layer (FTL). The buffer sends write requests to FTL when it is full or a flush request is received. FTL writes the data to flash memory after determining the physical location of the write requests. When FTL receives unaligned overwrite requests, it incurs read-modify-write operations.

WAFlash achieves the elimination of the side-effect of unaligned writes with three key techniques: (1) page replacement considering partial pages vs. full pages and chip congestion, (2) multi-version data storage in flash memory, and (3) partial page compaction in flash memory. We will elaborate on each of them in the following sections.

A. Partial Page and Congestion-Aware Buffer Replacement

To reduce unaligned writes to flash memory, we devise a new buffer replacement algorithm, Partial page and Congestion-aware LRU (PC-LRU). We design the algorithm based on the observations that (1) when chip congestion happens the cost of writing partial pages is much higher than that of writing full pages and (2) partial pages are likely to be overwritten and converted to full pages.

We manage the write buffer as an LRU stack L. The layout of the buffer is shown in Figure 3. I/O requests enter the top of L. They are then pushed to the bottom of L, further dropped out of the write buffer. Overwrites are merged with the existing requests in the buffer and moved to the top of L. The write buffer manages both full pages (e.g., $LPN(2)$ and $LPN(5)$) and partial pages (e.g., $LPN(17)$ and $LPN(20)$) using different replacement policies considering chip idleness. The full pages are interleaved with partial pages in the write buffer. When the chip is idle, the page at the bottom of L will be replaced. However, when the chip is not idle (or chip congestion occurs), PC-LRU needs to replace the full page closest to the bottom of L. With this approach, we can reduce the number of read-after-write operations and promote the conversion of partial page to a full page by keeping the partial pages in the buffer for a longer period.

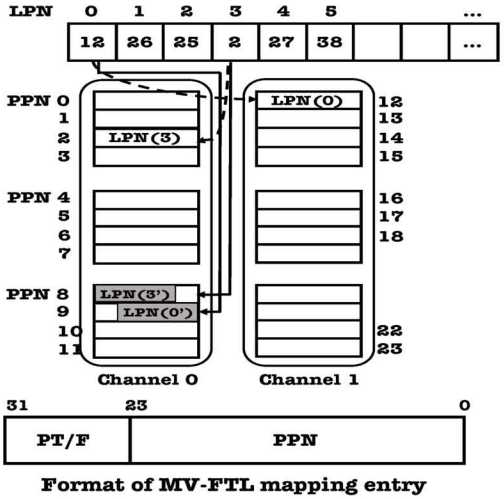


Fig. 4: Illustration of MV-FTL. LPN: logical page number; PPN: physical page number; PT: version pointer; F: entry flags. $LPN(3)$ has two versions stored in $PPN(2)$ and $PPN(8)$ respectively. $LPN(0)$ has two versions stored in $PPN(12)$ and $PPN(9)$.

B. Multi-Version Partial-Page Management

Because SSDs have limited RAM buffer capacity, flash memory has to serve unaligned writes when write buffers with PC-LRU cannot fully convert partial pages to full pages. To further eliminate read-modify-write operations from the critical I/O paths, WAFlash writes a partial page directly to flash memory as a new version of the page data without triggering the read-modify-write. Consequently, there could

be more than one valid physical pages that are mapped to the same logical page. Each of the physical pages stores a version of the data which corresponds to an unaligned write to the logical page. For accessing the multi-version partial-page aware flash memory, we design a new FTL scheme, called MV-FTL, which explores the flash pages that are not in use or the spare pages that are provided by SSD manufactures (e.g., Micron [16]).

A critical challenge of multi-version data management is that the size of FTL can be significantly increased for storing additional mapping entries for the newer version of data. To prevent increasing the size of FTL, we partition the existing mapping entries of FTL into two address areas: PPN and PT/F. PPN stores the physical page number. It uses the n least significant bits, where $page\_size$ * 2^n is equal to the SSD capacity. The remaining bits are used as the flags (F) or pointers (PT). If the entry stores a mapping of a full page, PT/F is set to all 0s. If data in a logical page has multiple versions, the mapping entry of the partial page uses PT/F to store the indexed location where to find its next version. PTs in multiple mapping entries may point to the same physical page for storing compacted small partial pages (see § III-C). The format of MV-FTL mapping entry is illustrated in Figure 4. For example, if we assume that the SSD capacity is 512 GB and page size is 16 KB, then PPN needs 25 bits. PT/F has 7 bits as shown in the figure. If the desired size of address space is smaller than the space that can be indexed using PT/F (e.g., $< 2^7$), we use direct indexing. Otherwise, we should use indirect indexing to increase the space of addressing using PT/F and increase the possibility of finding an idle physical page to store multi-version data.

Partial page writes: When an unaligned request is to write a significant portion (e.g., >50%) of a page $LPN(x)$, MV-FTL writes the partial page back to a new spare page ($Page_{new}$) in flash memory directly without reading the original page ($Page_{old}$). The distance between $Page_{new}$ and $Page_{old}$ is recorded in the mapping entry of $LPN(x)$. For both $Page_{old}$ and $Page_{new}$, we need to record which portion of the page is written. For fast comparison, we define partial-page write patterns as shown in Figure 5 with the assumption that page size is 16 KB and sector size is 4 KB. In $Pattern$ 0, the first 3 sectors in the page are written. In $Pattern$ 1, only sectors in the position of [1-3] are written. MV-FTL uses these defined patterns to determine how to serve a read request or merge two versions of the partial pages. For example, if a request is to read sectors [1-3] and $Page_{new}$ has $Pattern$ 1, then only $Page_{new}$ needs to be read. If a request is to read sectors [0-3] and $Page_{new}$ has $Pattern$ 1, then both $Page_{new}$ and $Page_{old}$ should be read to RAM for serving the request by merging sector [0] from $Page_{old}$ and sectors [1-3] from $Page_{new}$. To save space overhead, we store the pattern information in the OOB area of the corresponding physical pages. Typically, a flash page has 128 B OOB data [11], which is large enough to store the pattern information.

We use the example in Figure 4 for further illustration. $LPN(3)$ was mapped to $PPN(2)$ ($LPN(3)\rightarrow PPN(2)$).

Pattern 0	1	1	1	0
Pattern 1	0	1	1	1

Fig. 5: The unaligned write patterns. "1" indicates the sector is written; "0" indicates the sector is not written. We assume that page size is 16 KB and sector size is 4 KB. The partial page writes only access contiguous sectors and their sizes are larger than 50% of flash page size.

After receiving an unaligned write request to $LPN(3)$, MV-FTL writes the data back to $PPN(8)$. No additional read to $PPN(2)$ is needed. It is delayed until when the merging operation is executed. A new mapping entry ($LPN(3)\rightarrow PPN(8)$) is added to MV-FTL. After serving the unaligned write, there are two valid physical pages ($PPN(2)$ and $PPN(8)$) that are mapped to $LPN(3)$. The two physical pages are merged when GC is executed. Similarly, version 0 and version 1 of $LPN(0)$ are mapped to $PPN(12)$ and $PPN(9)$ respectively for reducing additional reads. The write pattern corresponding to the mapping entry of ($LPN(3)\rightarrow PPN(8)$) is $Pattern$ 0 while the pattern corresponding to the mapping entry of ($LPN(0)\rightarrow PPN(9)$) is $Pattern$ 1 according to which sectors that are written.

Full page writes: When a full page is written to $LPN(x)$, a new physical page is used to serve the request directly. All the previous versions of data in physical pages that were mapped to page $LPN(x)$ should be marked as invalid in MV-FTL and cleaned when GC is executed. **Page reads**: MV-FTL always uses the latest version of the physical pages to serve incoming read requests. If the requested data is not available according to the write pattern, it will issue another read for its earlier version of the data and then merge the two versions in RAM for serving the read requests. **Page merging and garbage collection**: Multiple versions of the data mapped to the same logical page number should be merged effectively to improve the space utilization and reduce the execution time of GC. Specifically, MV-FTL merges multiple version of the data with GC in two scenarios: (1) SSDs are idle or in a light load and (2) GC is triggered.

C. Compacted Partial Pages

If a request only writes to a small portion (e.g., ≤50%) of a logical page, serving it using one whole physical page wastes the space of flash memory when the physical page is mapped to a new version of the logical page. WAFLASH identifies and compacts multiple small partial pages and then stores them in a single physical page to both improve the page space utilization and further reduce the number of unaligned writes.

Destaging a batch of partial pages from write buffers: For creating compacted partial pages, PC-LRU destages a batch of unaligned writes to several small partial pages. It selects the partial pages that are adjacent in the logical page address space to explore space locality. For this purpose, we partition the space of logical page address into a sequence of regions. The region size should be significantly larger than the page size. WAFLASH only compacts the small partial pages

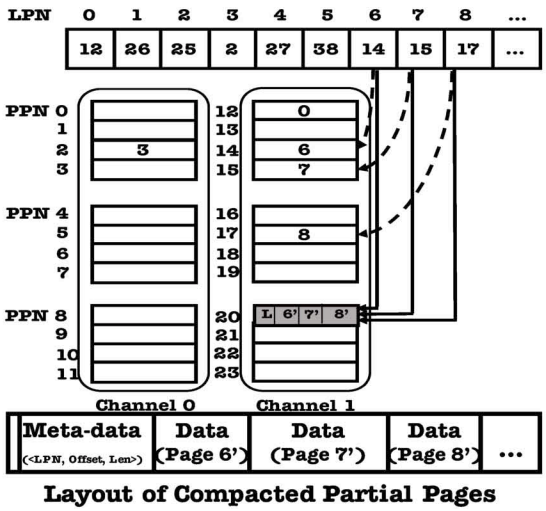


Fig. 6: Illustration of MV-FTL with compacted partial pages and its data layout. $LPN(6)$ has two versions stored in $PPN(14)$ and $PPN(20)$ respectively. Similarly, the two versions of $LPN(7)$ are stored in $PPN(15)$ and $PPN(20)$. And the two versions of $LPN(8)$ are stored in $PPN(17)$ and $PPN(20)$. The newer versions of $LPN(6)$, $LPN(7)$, $LPN(8)$ are stored in $PPN(20)$ as a compacted partial page.

that reside in the same region. As a result, it is possible that after one partial page is accessed the rest of them are to be accessed in a small time window.

Accessing the compacted partial pages: To create compacted partial pages, MV-FTL needs to find a spare flash page. The layout of the page is shown in Figure 6. If a physical page is used to store multiple compacted partial pages, its first two bytes should contain a unique identifier (e.g., $0xFFFF$). The rest of each physical page has a *meta-data* section and multiple *data* sections. The meta-data section is used by MV-FTL to index the partial pages in the physical page. The data section stores the data of a partial page. The meta-data section stores a list of triples $< LPN, offset, len >$, where LPN is the logical page number of the partial page, $offset$ is the offset of the partial page in the physical page, and len is the length of the partial pages in sectors.

Writing compacted partial pages: To update compacted partial pages, MV-FTL needs to prepare the data layout (meta-data) and then stores it together with the data in partial pages to the selected physical page. After writing the partial pages, MV-FTL updates the mapping entries with the new version pointer by setting PT. We use the example in Figure 6 for illustration. $LPN(6)$, $LPN(7)$, and $LPN(8)$ were mapped to $PPN(14)$, $PPN(15)$, and $PPN(17)$ respectively. Once PC-LRU destages the partial pages $LPN(6)$, $LPN(7)$, and $LPN(8)$ from the write buffer, MV-FTL stores the new version of $LPN(6)$, $LPN(7)$, and $LPN(8)$ to $PPN(20)$ with the specific data layout. Therefore, serving the 3 writes using the proposed compacted approach reduces the number of I/O requests from 3 pages writes plus 3 page reads to only 1 page write. For serving unaligned writes that overwrite the compacted data in $LPN(x)$, MV-FTL needs to read the data from PPN(y) which can be found using the index stored in PT in the mapping entry of $LPN(x)$. Then the data is merged with those from the incoming write requests. Finally, MV-FTL writes the latest merged version to another empty physical page $PPN(z)$ and updates PT to point to $PPN(z)$.

TABLE I: SSD characteristics (Micron [20]).

Parameter		Parameter	
SSD Capacity	48GB	#Pages/block	256
#Channels	4	Page size	16KB
#Chips/channel	4	Page read	20us
#Dies/chip	2	Page write	200us
#Planes/die	2	Page data transfer	51.2us
#Blocks/plane	4096	Block erase	1.5ms

Reading compacted partial pages: To read compacted partial pages, MV-FTL needs to read the latest version of the data stored in the physical pages that are mapped to the requested logical page. It can use the data layout to locate the requested page from the compacted partial pages. If the requested sectors is not available in the page, MV-FTL needs to read its earlier version of the data and merges the two versions of the data in RAM for serving the reads. **Page merging and GC** are triggered as explained in § III-B and executed using the data layout for indexing.

IV. WAFLASH IMPLEMENTATION

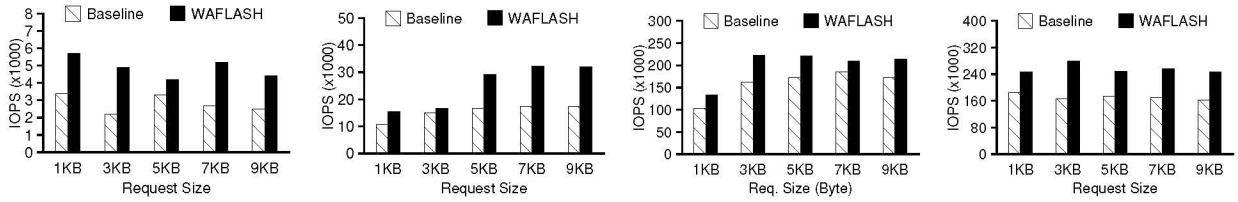
To run applications and real file systems, we implement WAFLASH on VSSIM, a QEMU/KVM-based SSD emulator that runs in real-time [10]. VSSIM emulates flash latencies on memory (RAMDisk) or fast SSDs. Its accuracy is validated against a commodity SSD [17]. VSSIM can efficiently model the SSD system with various design choices, such as hardware configurations (e.g., the number of channels, the block size, and the page size) and firmware schemes. The original write buffer in VSSIM is organized via a first-in-first-out (FIFO) circular queue. Instead of replacing a single page when the buffer is full, VSSIM evicts the data of one request, until all requests in the buffer are destaged to flash memory. The partial page-unconscious policy involves a lot of additional read operations. We make changes in the write buffer module to port the PC-LRU algorithm. We also integrate the multi-version storage and the page compaction technique in the FTL scheme. In all, we modify VSSIM with a total of 947 LOC.

We currently only implemented WAFLASH in page-level buffer schemes and page-level FTLs. We choose the page-level buffer schemes for their simplicity and efficiency; we focus on page-level FTLs for their best performance with decent mapping overheads if optimized mapping methods are used [18]. However, we believe our idea can also benefit other schemes. For example, we can port PC-LRU to block-level or hybrid-level buffer schemes by prioritizing destaging the buffer blocks with more full pages. If hybrid FTLs are deployed, we can apply MV-FTL to the log-block management which uses page-level address mapping [19]. We leave these as our future work.

V. EVALUATION

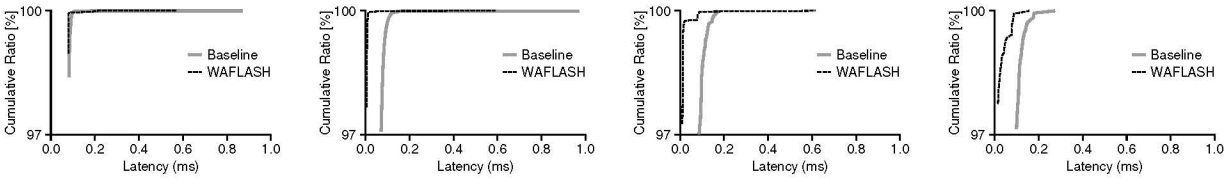
A. Experimental Methodology

Platforms and configurations: For a comprehensive study of the performance of WAFLASH, we emulated SSDs in



(a) Random Write (b) Fileserver (c) Webserver (d) Webproxy

Fig. 7: Filebench on the emulation-based platform.



(a) Random Write (b) Fileserver (c) Webserver (d) Webproxy

Fig. 8: Tail latencies. The figures show the CDF of request latencies in various workloads.

VSSIM. Its parameters are listed in Table I. Its sector size is 4 KB. We use a machine with 3.2 GHz 6-core Intel(R) Core(TM) i7-8700 and 64 GB DRAM. We use page-level FTL provided by VSSIM as the baseline on this platform.

Workloads characteristics: For the whole-system evaluation, we use Filebench with four personalities [21]: one micro workload (Random Write) and three macro workloads (Fileserver, Webserver, and Webproxy). The characteristics of these workloads are publicly reported in [22], [23] and described in § V-B.

B. Benchmark-Driven Evaluations on VSSIM

In this subsection, we evaluate WAFLASH on the emulation platform with Filebench [21]. We increase I/O request sizes of the benchmarks from 1 KB to 9 KB in the experiments. We run four workloads from Filebench. The *random write* workload was set to use a large file of 5 GB and three threads to perform random writes to the file. For the *fileserver* workload, the number of files is 10000, the number of threads is 50, and the mean file size is 128 KB. For the *Webserver* workload, the number of files is 1000, the number of threads is 100, and the mean file size is 16 KB. For the *Webproxy* workload, the number of files is 10000, the mean file size is 128 KB, and there are 100 threads. Figure 7 shows the I/O throughput across the four workloads. The results include kernel, file-system, and QEMU overhead in addition to device-level latencies.

We have the following observations. (1) With WAFLASH, the I/O throughput of WAFLASH is increased by up to 125%, 86%, 37%, and 68% respectively. Among the four workloads, the *random write* workload achieves the most improvement because it has the highest unaligned write ratio and worst spatial locality. (2) We count the percentage of partial pages in the buffer when WAFLASH is enabled. PC-LRU has 21% more partial pages in the buffer compared to Baseline for all the workloads. This is because PC-LRU prioritizes the destaging of full pages and keeps the partial pages in the buffer for a longer time. As a result, less partial pages can be written to the flash memory and these pages have more chance to become full pages, which are the desired flash access patterns. (3) Compared to Baseline, PC-LRU decreases 4.5–10.6% partial pages written to flash for all the workloads. The reason is that there are some partial pages merged to full pages due to their residence in the buffer and the full page replacement leads to a fewer number of buffer eviction for serving incoming requests. As serving partial-page writes is much more expensive than serving full-page writes, the reduced number of partial-page writes to flash helps system performance. (4) Partial page compaction decreases 7–12% write operations to flash memory. This approach not only improves the page space utilization but also reduces request latency due to the reduced number of write requests.

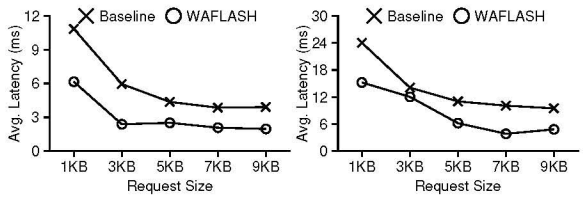
CDF latencies: Figure 8 shows the CDF of request latencies from the same experiments. From this figure, one can find that for all workloads, the Baseline approach has longest tail latencies and WAFLASH exhibits the shortest latencies. In addition, for some evaluated workloads, the 99th percentile performance gains achieved by WAFLASH is higher than average values.

C. Impact on Read Performance

So far we have shown that WAFLASH improves the overall I/O performance. One concern is that WAFLASH may hurt read latency, which is generally more critical in storage performance. WAFLASH can reduce read latency while improving write performance. To verify this, we measures the average read and write latency of the *fileserver* workload in Figure 8b (other workloads show similar results). As Figure 9 shows, the read and write latency are reduced by up to 60% and 62% respectively. This is because WAFLASH reduces the number of read and write operations in flash channels in the critical I/O path, which benefits both read and write operations. The overhead of multi-version FTL is well amortized over the running of the workload.

D. Overhead Analysis

Multi-version management overhead: Instead of invalidating the old version of the data, MV-FTL keeps both of the copies valid until they are merged. However, due to the



(a) Read (b) Write

Fig. 9: The read and write latency of *fileserver*.

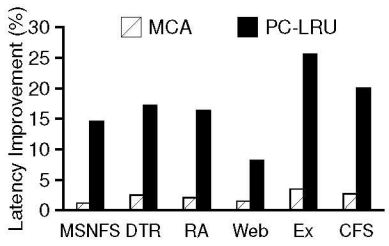


Fig. 10: WAFLASH vs. MCA.

out-of-place update nature of flash, MV-FTL does not require additional space for keeping the two versions of data of the same logical page. It only changes the state transition of pages and meta-data management. As discussed in § III-B, we control the additional meta-data overhead to be acceptable by partitioning existing mapping entries in FTL with new fields and using direct and indirect indexing techniques to reduce the overhead.

Page compaction overhead: Page compaction increases flash page utilization with little space overhead. The source of overhead is mainly from storing the triple $< LPN, offset, len >$ for indexing the partial pages in the physical page. If we assume that the page size is 8 KB and the SSD size is 512 GB, the index triple requires 6 B for each partial page. We can store the index in the physical page or in the out-of-band area of the page. Then the space overhead for meta-data management is about 0.15% in the worst scenario. The space overhead can be significantly decreased as the page size is increased for larger capacity SSDs and as the fraction of full-page writes increases in real workloads.

E. WAFLASH *vs. the State-of-the-Art: MCA*

As mentioned previously, MCA represents the best algorithm in write buffer management considering side-effects of unaligned writes [7]. It delays evicting the partial pages that cause chip waiting. We implement MCA and PC-LRU in SSDSim based on existing literature [7]. In the experiments, we use 6 real block-level traces because MCA was evaluated using I/O traces. Specifically, they include Microsoft Production Servers (MSNFS, DTR, RA, and CFS), Microsoft Windows Servers (Ex), and Florida International University (Web) [12]. Figure 10 compares PC-LRU and MCA in terms of their latency improvements against the Baseline. As shown, PC-LRU significantly outperforms MCA for all the workloads: PC-LRU has 25.6% performance improvements while MCA only reduces request latencies by 2.7%.

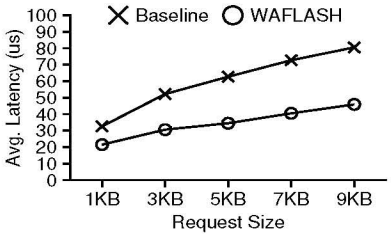


Fig. 11: Filebench on the Cosmos+ OpenSSD platform.

F. Benchmark-Driven Evaluations on OpenSSD

Finally, we evaluate WAFLASH on the Cosmos+ OpenSSD board [24]. The development PC runs Ubuntu 14.04 and Xilinx software. Xilinx Vivado suite allows us to change the configuration of the board. We use the HDF files provided by the vendor. Using Xilinx SDK, we change the firmware and program the FPGA board. The flash capacity is 1 TB. The flash page size is set to 16 KB and the sector size is 4 KB. We use the GreedyFTL algorithm, which is a page-mapping FTL with greedy GC, as the baseline.

We use the Random Write workload in Filebench. Figure 11 shows the average I/O latency. Compared to the baseline system, WAFLASH reduces the average latency by up to 43% for random write. For I/O throughput, WAFLASH delivers similar behaviors to those observed on the VSSIM platform.

VI. RELATED WORK

Buffer management schemes: Many approaches were proposed to utilize RAM buffers to improve the performance of slow storage devices. Earlier schemes are designed for HDDs by exploiting spatial and temporal locality [25], [26]. These approaches can be applied to SSDs but with sub-optimal performance. Later, flash-aware schemes are developed for SSDs considering flash characteristics. These efforts include page-level schemes (e.g., MCA [7], CFLRU [27], and FOR [28]), block-level schemes [15], [29], and hybrid schemes [30], [31].

Among these designs, MCA [7] is the only partial page-aware scheme that is mostly related to WAFLASH. We have discussed the limitation of MCA in § V-E. MCA only delays the eviction of partial pages that cause chip waiting due to additional reads while PC-LRU delays the eviction of all partial pages. WAFLASH can further reduce the number of requests on channels by replacing more full pages. In addition, compared to MCA, WAFLASH keeps partial pages in the buffer for a longer time, so that more full pages can be generated. Finally, WAFLASH uses MV-FTL to further reduce unaligned writes when the efficiency of write buffer is limited. Researchers have also studied the buffer management across multiple SSDs [32] and addressed unaligned accesses on multiple HDDs [33].

Flash translating layer techniques: A number of FTL-level approaches have been proposed to improve SSD performance by reducing GC overhead or increasing I/O parallelism. According to the granularity of address mapping, existing FTLs schemes can be classified into *page-level* schemes [18], [34], *block-level* schemes, and *hybrid* schemes [19], [35].

While the above schemes aim to reduce copy, write, and erase operations, none of them focus on unaligned writes. MV-FTL aims to reduce additional reads and the number of partial-page writes caused by unaligned writes by utilizing multi-version data storage and partial-page compaction. Importantly, WAFLASH is orthogonal to and can further improve the above FTL techniques. Besides the FTL optimization, WAFLASH exploits the novel buffer scheme to alleviate the side-effects of unaligned writes from the upper layer.

VII. Conclusion

SSDs become a norm in storage systems, but applications and system software may ignore the hardware complexity inside SSDs. Unaligned writes, which are common in practical I/O workloads, can significantly decrease SSD performance. We have proposed and implemented a cooperative mechanism, named WAFLASH, to comprehensively reduce almost all the side-effects of the unaligned writes in SSDs. WAFLASH delays the eviction of partial pages in write buffers, stores multi-version data in flash memory, and compacts multiple partial pages to a flash page, to circumvent write amplification and additional reads in the critical I/O path. Using the Filebench benchmark we show that WAFLASH provides up to 125% performance improvement.

Acknowledgment

This research was supported by US National Science Foundation under CNS 1906541. This work was also supported in part by the National Key Research and Development Program of China No.2021ZD0110700, the National Science Foundation of China No. 62172361, and the Program of Zhejiang Province Science and Technology No.2022C01044.

References

[1] Fusion-io, "Fusion-io ioDrive," https://www.sandisk.com/business/datacenter/products/flash-devices/pcie-flash/sx350 .

[2] "Report: SSD Market Doubles, Optical Drive Shipment Rapidly Down," http://www.myce.com/news/report-ssd-market-doubles-optical-drive-shipment-rapidly-down-70415, 2014.

[3] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proc. ASPLOS'09*, 2009.

[4] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments," in *Proc. of FAST'16*, 2016.

[5] L. Harbaugh, "How Storage Solutions Are Rising to Meet Edge Computing Needs," https://insights.samsung.com/2018/05/09/how-storage-solutions-are-rising-to-meet-edge-computing-needs/, 2018.

[6] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "Integrating lsm trees with multi-chip flash translation layer for write-efficient kvssds," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 1, pp. 87–100, 2020.

[7] J. Seol, H. Shim, J. Kim, and S. Maeng, "A Buffer Replacement Algorithm Exploiting Multi-Chip Parallelism in Solid State Disks," in *Proc. of CASES'09*, 2009.

[8] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting Intra-Request Slack to Improve SSD Performance," in *Proc. of ASPLOS'17*, 2017.

[9] Y. Lu, J. Shu, and W. Zheng, "Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems," in *Proc. of FAST'13*, 2013.

[10] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "VSSIM: Virtual machine based SSD simulator," in *Proc. of MSST'13*, 2013.

[11] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proc. of ATC'08*, 2008.

[12] "SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis," http://iotta.snia.org.

[13] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/o characteristics of smartphone applications and their implications for emmc design," in *Proc. of IISWC'15*, 2015.

[14] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity," in *Proc. of ICS'11*, 2011.

[15] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proc. of FAST'08*, 2008.

[16] "NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product," https://www.micron.com/resource-details/fea5cfd9-ee93-47f4-b2af-cd494d3291c3.

[17] Intel Corporation, "Intel X25-M SATA Solid-State Drive Specification," http://download.intel.com/design/flash/nand /mainstream/mainstream-sata-ssd-datasheet.pdf.

[18] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proc. of ASPLOS'09*, 2009.

[19] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, 2007.

[20] Micron Technology, Inc., "NAND Flash Memory MLC Data Sheet, MT29E512G08CMCCBH7-6 NAND Flash Memory," http://www.micron.com/.

[21] "Filebench," http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[22] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-Volatile Memory," in *Proc. of FAST'13*, 2013.

[23] Y. Lu, J. Shu, and W. Wang, "ReconFS: A Reconstructable File System on Flash Storage," in *Proc. of FAST'14*, 2014.

[24] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ openssd: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, Jul. 2020. [Online]. Available: https://doi.org/10.1145/3385073

[25] F. J. Corbato, "A Paging Experiment with the Multics System," Report MIT Project MAC Report MAC-M-384, 1968.

[26] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality," in *Proc. of FAST'05*, 2005.

[27] 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 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006, pp. 234–241.

[28] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-Aware Buffer Management in Flash-Based Systems," in *Proc. of SIGMOD'11*, 2011.

[29] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," *IEEE Transactions on Consumer Electronics (TCE)*, vol. 52, no. 2, pp. 485–493, 2006.

[30] G. Wu, B. Eckart, and X. He, "BPAC: An Adaptive Write Buffer Management Scheme for Flash-Based Solid State Drives," in *Proc. of MSST'10*, 2010.

[31] Q. Wei, C. Chen, and J. Yang, "Cbm: A cooperative buffer management for ssd," in *Proc. of MSST'14*, 2014.

[32] K. Ganesh, Y. Kim, M. Debnath, S. Park, and J. Lee, "LAWC: Optimizing Write Cache using Layout-Aware I/O Scheduling for All Flash Storage," *IEEE Transactions on Computers (TC)*, vol. 66, pp. 1890–1902, 2017.

[33] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving Unaligned Parallel File Access with Solid-State Drives," in *Proc. of IPDPS'13*, 2013.

[34] M. Chiang, P. C. Lee, and R. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software: Practice and Experience (SPE)*, vol. 29, no. 3, pp. 267–290, 1999.

[35] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for CompactFlash Systems," *IEEE Transactions on Consumer Electronics (TCE)*, vol. 48, no. 2, pp. 366–375, 2002.