WAFLASH: Taming Unaligned Writes in Solid-State Disks

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 the cloud computing. Due to the architectural nature of SSDs, performance-sensitive applications may generate a large amount of unaligned writes on SSDs, which can cause many issues (e.g., chip congestion, sub-request blocking, chip load imbalance, and write space amplification).

We present the design and implementation of 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–prioritizing eviction of fully-filled pages over partially-filled pages in write buffers, storing multi-version data requested by unaligned writes/overwrites in flash memory, and 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 demonstrate that the advantages of WAFLASH using trace-driven simulations in SSDSim and experiments with a prototype implementation in VSSIM. For 6 real I/O workloads, WAFLASH obtains average latency improvement by up to 15.1%, 31.3%, and 39.5% for 4 KB, 8 KB, and 16 KB page size, respectively. Across 4 Filebench workloads, WAFLASH improves the benchmark throughput by up to 109.6%.

1 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 [11]. 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 [54], and flash/SSD-based storage becomes a norm in various storage systems from high-end servers in data centers [4, 15] to mobile computers on the edges of the cloud computing [16].

Although other interfaces exist, most of the existing systems use the legacy block interface to access SSDs for portability and compatibility with HDDs. However, the smallest read/write unit is a *page* (e.g., 4 KB) in SSDs, instead of a *sector* (e.g., 512 B) in HDDs. With this architectural nature, applications may generate a large number of writes that are not aligned with the page boundary of SSDs. We call them *unaligned writes* in this paper. As shown in § 2.2, the unaligned writes are common in practical I/O workloads. With the increase of flash density, the page size is also increased from smaller than 1 KB to 16 KB [31, 55]. This makes the unaligned problem more severe because existing systems access SSDs without knowing the change in page sizes.

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 [52]. 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 [9]. 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 is the state-of-the-art scheme that focuses on reducing chip waiting with new buffer management [52]. Slacker can alleviate load imbalance across chips by considering the completion time of

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

each sub-request of a large request in I/O scheduling [9]. Lu et al. used object interface to replace the block interface for SSD accesses [39]. 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 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 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 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. 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 in SSDSim [19] for trace-driven simulations and in QEMU/KVM-based platform based on VSSIM [62] 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 39.5% and 109.6%, respectively.

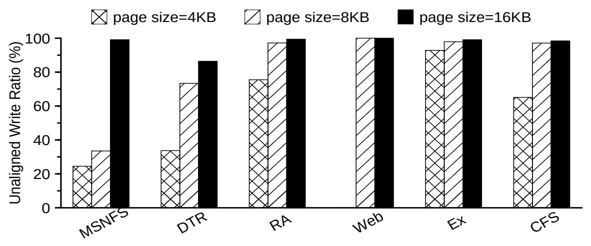


Figure 1: **Percentages of unaligned writes in workloads.**

2 Background and Motivation

2.1 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 metadata area (called out-of-band (OOB) data) [2]. 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 and 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), which has a smaller size than a flash page. 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.

2.2 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. Figure 1 shows the ratio of unaligned writes in the set of block-level traces from system software (e.g., MSNFS, CFS) [21] and web applications (e.g., Web) [21] when the page size is 4 KB, 8 KB, and 16 KB. The average ratio of unaligned writes is 58.3% except for *Web* whose requests are all aligned with a 4 KB page size. Even worse, the trend of increasing page size of flash memory can further increase the ratio of unaligned writes. For example, when the page size is increased from 4 KB to 16 KB, the ratio of unaligned writes in the MSNFS trace will increase from 24.5% to 99.1% and the average ratio for all workloads reaches to 97.2%.

To experimentally investigate the effects of unaligned writes on flash read/write performance, we conduct exper-

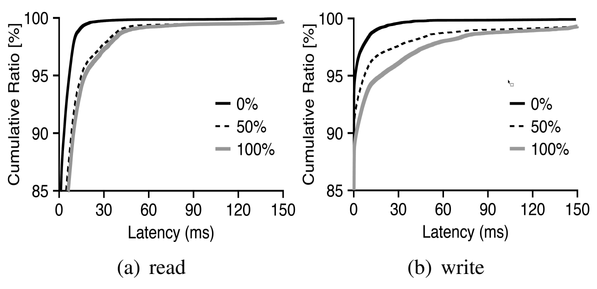


Figure 2: **The impact of serving unaligned writes on SSD performance.** Both read and write latency become longer.

iments on a simulated SSD [19] 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. All the requests have an 8 KB size and 30% of them are reads and the rest are writes. All the reads are aligned with page boundaries. But for writes, we make part of them unaligned by setting their offsets as 1 KB to page boundaries. We vary the ratio of unaligned writes from 0% to 100%. Figure 2(a) and 2(b) 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} percentiles, respectively. Therefore, it is necessary to eliminate the side-effects of them to improve SSD performance.

3 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 3 shows the general system architecture considered in this paper. 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 [29]. 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 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 each of them in the following sections.

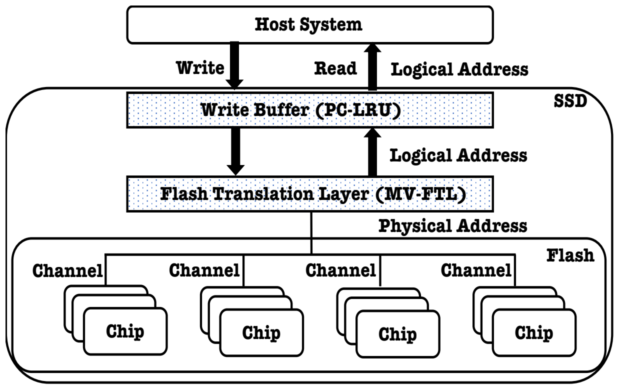


Figure 3: **System architecture.** The proposed write buffer management scheme (PC-LRU) and multi-version-aware FTL (MV-FTL) are applied to the write buffer and FTL layer inside SSDs respectively.

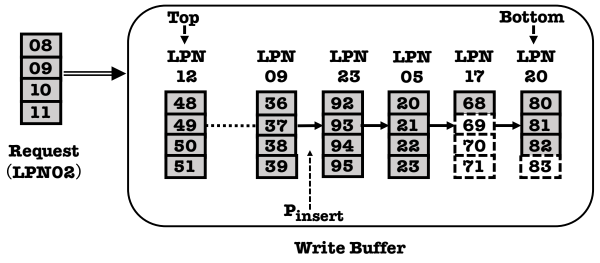


Figure 4: **PC-LRU write buffer management.** *LPN*: logical page number. We assume 1 page consists of 4 sectors. *LPN*(02), *LPN*(12), *LPN*(09), *LPN*(23), and *LPN*(05) are full pages and *LPN*(17) and *LPN*(20) are partial pages. When chips are idle, upon serving the write to the full page *LPN*(02), 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*(05) is replaced.

3.1 Partial Page and Congestion-Aware Buffer Replacement Scheme

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 4. 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 request in the buffer and moved to the top of *L*. The write buffer manages both full pages (e.g., *LPN*(02) and *LPN*(05)) and partial pages (e.g., *LPN*(17) and *LPN*(20)) using different replacement policies considering chip idleness. 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 a full page to reduce read-after-write operations. In this scenario, if a full page is at the bottom of L, it will be replaced directly. If a full page is not at the bottom and there are partial pages between the position of the full page and the bottom of L, each of the partial pages should be re-inserted to a new position P_{insert} in the stack.

For promoting the conversion of partial pages to full pages, P_{insert} is adaptively adjusted based on the likelihood that a partial page can be converted to a full page by staying in the buffer for a certain period of time. We call this time period, *page conversion distance* (PCD). The PCD of a partial page X is defined as the total number of distinct pages between two consecutive references to the page X. We track the PCDs of the partial pages in write buffers and compute average conversion distance (ACD) as the average value of all PCDs tracked online. PC-LRU updates ACD after each occurrence of overwrites to a partial page. We then set the distance between P_{insert} and the bottom of L to ACD.

PC-LRU is adaptive to the access patterns of workloads. For example, if no unaligned requests arrive, all the space of a write buffer will be used for serving aligned requests just following the behavior of the original LRU algorithm. In contrast, when the ratio of unaligned writes is increased, more space of write buffers will be allocated for serving the requests when the chip congestion is detected. We used the LRU algorithm in the prototyped system because it is the most widely-used buffer cache replacement algorithm in flash [19, 62]. In the future work, we plan to apply the principle to other hit-ratio based page replacement algorithms (e.g., ARC [42] and LIRS [24]).

3.2 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 [56]).

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 § 3.3). The format of MV-FTL mapping entry is illustrated in Figure 5. For example, if we assume that the SSD capacity is 256 GB and page size is 4 KB, then PPN needs 26 bits. PT/F has 6 bits as shown in the figure. If the desired size of address space is smaller than the space provided by PT/F (e.g., $< 2^6$), 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.

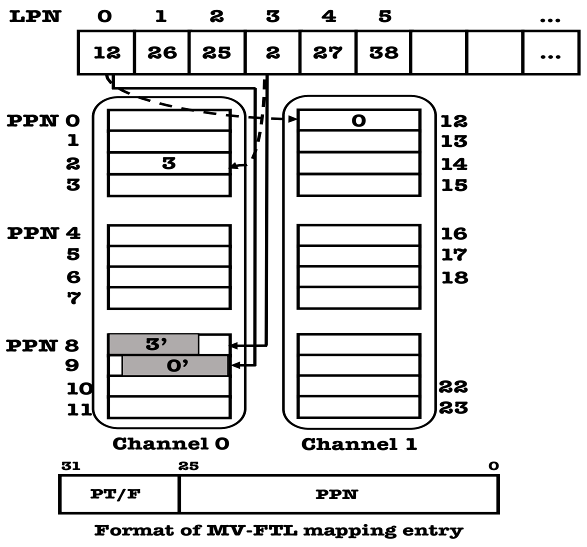


Figure 5: **Illustration of MV-FTL.** LPN: logical page number; PPN: physical page number; PT: version pointer; F: entry flags. $LPN(3)$ has two versions stored in $PPN(2)$ and $PPN(8)$ respectively. $LPN(0)$ has two versions stored in $PPN(12)$ and $PPN(9)$.

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 9 partial-page

Pattern 0	1	1	1	1	1	1	1	0
Pattern 1	0	1	1	1	1	1	1	1
Pattern 2	1	1	1	1	1	1	0	0
Pattern 3	0	1	1	1	1	1	1	0
Pattern 4	0	0	1	1	1	1	1	1
Pattern 5	1	1	1	1	1	0	0	0
Pattern 6	0	1	1	1	1	1	0	0
Pattern 7	0	0	1	1	1	1	1	0
Pattern 8	0	0	0	1	1	1	1	1

Figure 6: **The 9 unaligned write patterns.** Page size is 4 KB and sector size is 512 B. "1" indicates the sector is written; "0" indicates the sector is not written.

write patterns as shown in Figure 6 with the assumption that page size is 4 KB and sector size is 512 B. In *Pattern* 0, the first 7 sectors in the page are written. In *Pattern* 3, only sectors in the position of [1-6] 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* 3, then only $Page_{new}$ needs to be read. If a request is to read sectors [0-3] and $Page_{new}$ has *Pattern* 3, 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 4 KB page has 128 B OOB data [2], which is large enough to store the pattern information.

We use the example in Figure 5 for further illustration. $LPN(3)$ was mapped to $PPN(2)$ ($LPN(3) \rightarrow PPN(2)$). 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* 6 while the pattern corresponding to the mapping entry of ($LPN(0) \rightarrow PPN(9)$) is *Pattern* 2 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.

3.3 Compacted Partial Pages

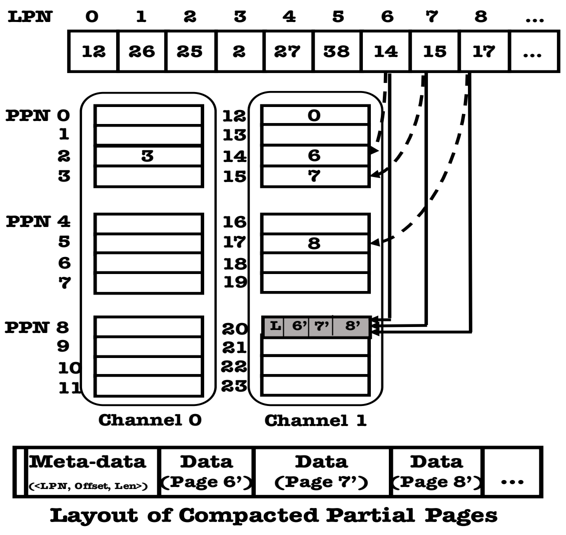


Figure 7: **Illustration of MV-FTL with compacted partial pages and its data layout.** $LPN(6)$ has two versions stored in $PPN(14)$ and $PPN(20)$ respectively. Similarly, the two versions of $LPN(7)$ are stored in $PPN(15)$ and $PPN(20)$. And the two versions of $LPN(8)$ are stored in $PPN(17)$ and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

If a request only writes to a small portion (e.g., $\leq$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 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 7. 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 7 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 the $PPN(20)$ 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 $PPN(z)$.

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 § 3.2 and executed using the data layout for indexing.

4 WAFLASH Implementation

We implement WAFLASH on the following two platforms: a trace-driven simulator and a virtual machine based emulator.

SSDSim-based implementation. To facilitate accurate analysis of SSD behavior at the device level, we first implement WAFLASH based on a recently-popular trace-driven simulator SSDSim [19]. Its accuracy has been validated over a real hardware prototyping. We choose SSDSim because it is clean-slate and it has detailed implementations of a page-level buffer management and FTL schemes. SSDSim also provides inter-flash and intra-flash parallelism by channel-level, chip-level, die-level, and plane-level parallel accesses.

Table 1: **SSD characteristics** (Micron [43]).

Parameter		Parameter	
SSD Capacity	256 GB	#Pages/block	256
#Channels	4	Page size	4/8/16KB
#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

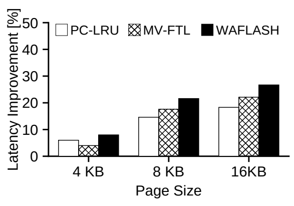
The vanilla system has 6844 lines of code (LOC). It provides a LRU-based page-level buffer replacement scheme and a page-level address mapping and regular data storage for unaligned writes. To enable the proposed approaches, we integrate all the features of WAFLASH by adding 1035 LOC using the C language.

VSSIM-based implementation: To measure application performance on host systems and SSD behavior, we also implement WAFLASH on VSSIM, a QEMU/KVM-based SSD emulator that runs in real-time [62]. VSSIM emulates flash latencies on memory (RAMDisk) or fast SSDs. Its accuracy is validated against a commodity Intel X25M SSD [20]. 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.

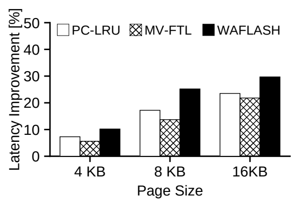
The original write buffer in VSSIM is organized via a first-in-first-out (FIFO) circular queue. Instead of replacing a single page when the buffer is full, VSSIM evicts the data of one request, until all data in the buffer are destaged to flash memory. The partial page-unconscious policy involves a lot 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.

5 Evaluation

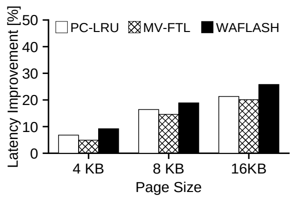
We present extensive evaluations to demonstrate the behaviors of WAFLASH. We address the following questions. (i) How well does WAFLASH perform under various workloads? (ii) Why is WAFLASH effective? (iii) When is WAFLASH not effective? (iv) How is WAFLASH affected by its parameters? (v) What is the overhead that WAFLASH incurs?



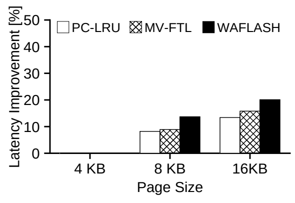
(a) MSNFS



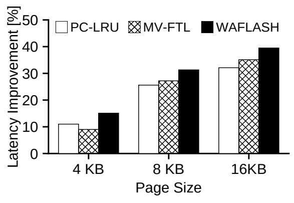
(b) DTR



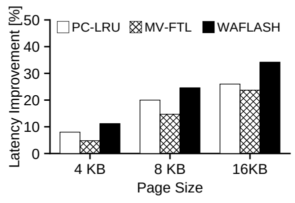
(c) RA



(d) Web

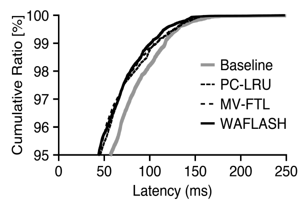


(e) Ex

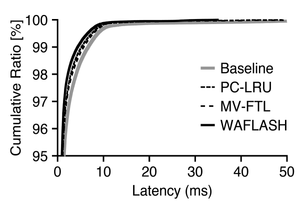


(f) CFS

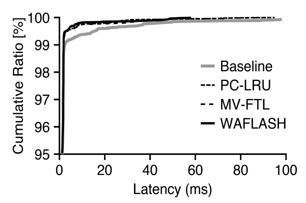
Figure 8: **Average latency improvements compared to the Baseline.** The page sizes are varied from 4 KB to 16 KB.



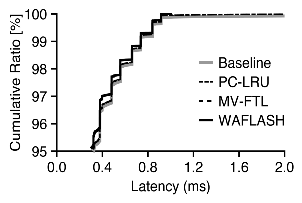
(a) MSNFS



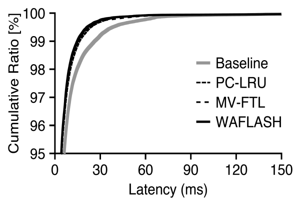
(b) DTR



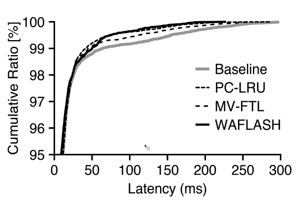
(c) RA



(d) Web



(e) Ex



(f) CFS

Figure 9: **Tail latencies.** The figures show the CDF of request latencies in various workloads. The page size is 8 KB.

5.1 Experimental Methodology

Platforms and configurations: We evaluate WAFLASH on two implementations as described in § 4. For the first implementation, we model a 256 GB SSD with related parameters listed in Table 1. The host interface and the channel interface are implemented based on NVM-e [1] and ONFi 3.1 specification [46] respectively. The simulated SSD uses a most-optimum page-level FTL scheme. For the second platform, the emulated SSD has the same parameters as the first platform except that its capacity is 48 GB due to the limitation of the machine's DRAM. We use a machine with 3.2 GHz 6-core Intel(R) Core(TM) i7-8700 and 64 GB DRAM.

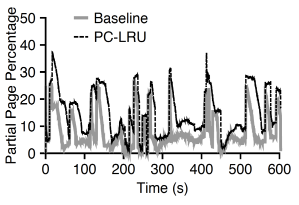
Workloads characteristics: For the simulation platform, we use 6 real block-level traces from Microsoft Production Servers (MSNFS, DTR, RA, and CFS), Microsoft Windows Servers (Ex), and Florida International University (Web) [21] to evaluate WAFLASH. These traces cover a diverse set of workloads. The detailed characteristics of these traces on unaligned writes are summarized in Figure 1. For the emulation platform, we choose Filebench with 4 personalities [10]: 1 micro workload (Random Write) and 3 macro workloads (Fileserver, Webserver, and Webproxy). The characteristics of these workloads are publicly reported in [33, 38].

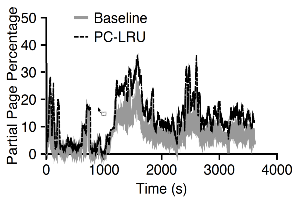
5.2 Trace-Driven Evaluations

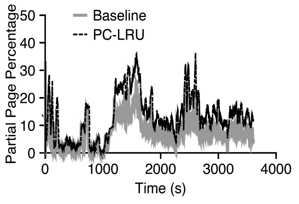
5.2.1 Major Results

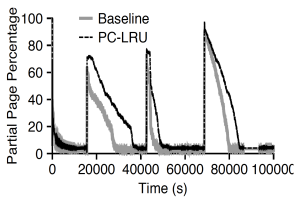
Average latencies: Figure 8 plots the average latency improvements under various page sizes. We include request latencies of all reads and writes as all of them can be affected by unaligned writes (see §2.2). The Baseline approach, which refers to the default SSDSim with LRU buffer management and the most-optimum FTL [19] and without multiple-version data storage, has long average latencies. In contrast, as new WAFLASH features are added into SSD: PC-LRU (§3.1), MV-FTL(§3.2+§3.3), we obtain considerable performance improvements. When all features are added together (WAFLASH=PC-LRU+MV-FTL), the average latencies can be further improved by 8–15.1%, 13.7–31.3%, and 13.4–39.5% over Baseline, for 4 KB, 8KB, and 16KB page sizes, respectively. Note that for Web, WAFLASH is not effective at a page size of 4 KB because all requests are aligned with the page boundaries. However, as the page size changes to 8 KB and 16 KB, Web can also benefit from WAFLASH because 100% of the write requests are unaligned writes. We also observe that as the page size increases, the improvement ratio becomes larger because there are often more unaligned write requests.

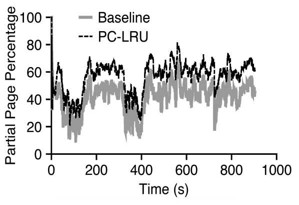
CDF latencies: Figure 9 shows the CDF of request la-


(a) MSNFS


(b) DTR


(c) RA


(d) Web


(e) Ex

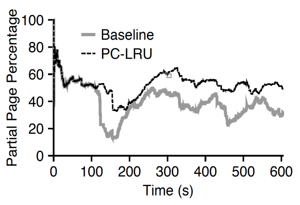

(f) CFS

Figure 10: **The partial page ratio in the buffer.** X-axis shows the running time.

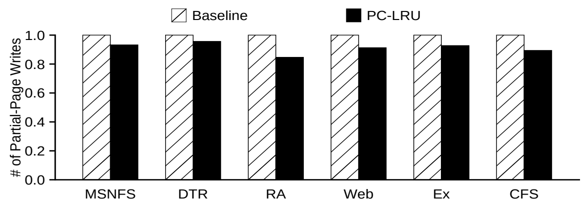


Figure 11: **The number of partial-page writes to Flash.**

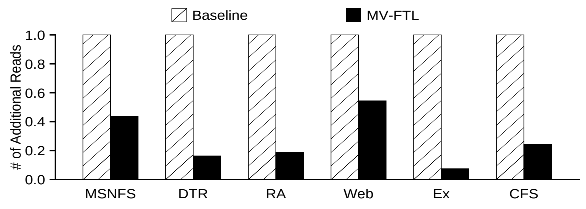


Figure 12: **The number of additional reads to Flash.**

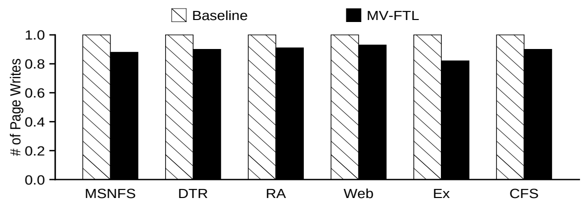


Figure 13: **The number of writes to Flash.**

tencies from the same experiments at the page size of 8 KB (other sizes have similar behaviors). 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 95th percentile performance gains achieved by WAFLASH is higher than average values. This specifically happens for workloads such as MSNFS and DTR, with 3.7% and 10.4% higher improvement respectively in 95th percentile analysis, compared to the average latency results in the above-mentioned subsection.

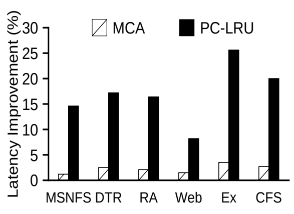
5.2.2 Why WAFLASH is Effective?

Numbers of partial pages in PC-LRU: To show what is happening inside the SSD, we count the percentage of partial pages in the buffer when WAFLASH is enabled at the page size of 8 KB. As shown in Figure 10, PC-LRU has 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.

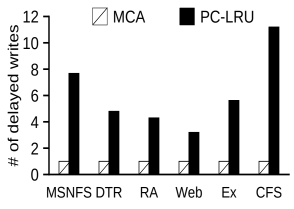
Numbers of partial pages written to flash: To confirm this, Figure 11 plots the number of partial pages written to flash in FTL in the above experiment. Compared to Baseline, PC-LRU decreases 4.5–10.6% partial pages written to flash for all the workloads. The reason is that there are some partial pages 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 (§1), the reduced number of partial-page writes to flash helps system performance.

The number of additional reads in MV-FTL: Besides the efforts in buffer layer, MV-FTL can further alleviate the side-effects of unaligned writes. To have a very accurate picture of WAFLASH's benefits, Figure 12 plots the number of additional reads on the flash in the previous experiment. Compared to Baseline, MV-FTL decreases 45.5–92.5% additional reads for all the workloads. This not only shortens latencies of unaligned writes, but also improves latencies of other requests because the chips are less congested. Another observation is MV-FTL did not completely eliminate the additional reads because it may trigger additional reads when reading a logical page stored in multiple physical pages. However, the performance benefit of MV-FTL significantly offsets the cost of serving such reads.

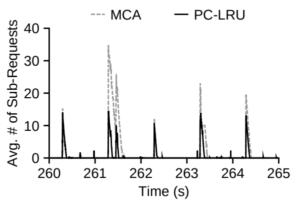
The number of writes in MV-FTL: Figure 13 plots the number of page writes to the flash when partial page compaction is also enabled in MV-FTL. Partial page compaction decreases 7–12% write operations to flash memory for all the



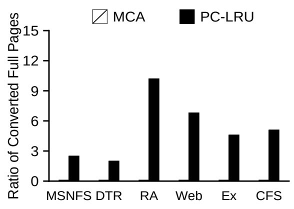
(a) The average latency improvement against Baseline



(b) The number of delayed partial pages in the buffer



(c) The average number of sub-requests on the channels in the CFS workload.



(d) The ratio of full pages converted from partial pages.

Figure 14: **WAFLASH vs. MCA**.

workloads. This approach not only improves the page space utilization but also reduces request latency due to the reduced number of write requests.

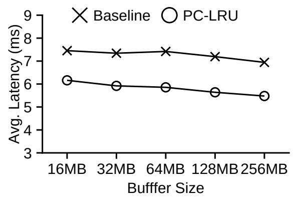
5.3 WAFLASH vs. the State-of-the-Art: MCA

As mentioned previously, MCA represents the state-of-the-art algorithm in write buffer management considering side-effects of unaligned writes [52]. It delays evicting the partial pages that cause chip waiting. We implement MCA in SSD-Sim based on existing literature [52]. If a new write request arrives when the buffer is full, MCA will preferentially select a full or partial page that avoids chip-waiting as a victim to serve the incoming requests.

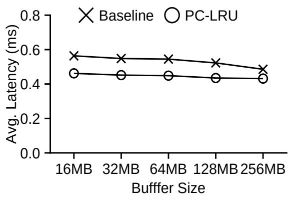
Figure 14(a) compares PC-LRU and MCA in terms of their latency improvements against the Baseline. As shown, PC-LRU significantly outperforms MCA for all the workloads: while PC-LRU has 8.2-25.6% performance improvements, MCA only reduces request latencies by 1.2-2.7%. There are two reasons for PC-LRU achieving better performance than MCA.

First, 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 when chip congestion happens. Therefore, PC-LRU is more aggressive in replacing full pages to reduce the total number of requests being served in flash in this scenario. As plotted in Figure 14(b), PC-LRU delays the eviction of 3.3–11.2X more partial pages in the buffer than MCA, thus it can largely reduce the number of sub-requests on channels (see Figure 14(c)).

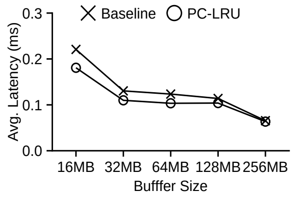
Second, PC-LRU makes partial pages to stay longer in the buffer by utilizing PCD, so that they have more chance to become full pages, which are the desired accesses for flash. In contrast, MCA only shifts one position in the LRU list to delay the eviction of the partial page, thus this partial page may be quickly destaged from the buffer if it does not cause chip waiting anymore in the future. As shown in Figure 14(d), although MCA can hardly convert partial pages to full pages in the buffer, PC-LRU can do this with a much higher ratio.



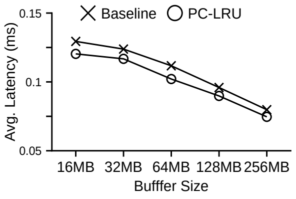
(a) MSNFS



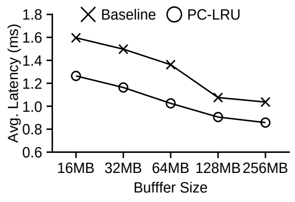
(b) DTR



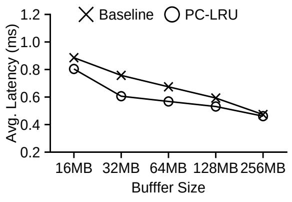
(c) RA



(d) Web



(e) Ex



(f) CFS

Figure 15: **Buffer size impact on average latency.**

Besides the efforts in write buffers, WAFLASH reduces the unaligned writes to flash with the multi-version data storage and partial page compacting techniques when the buffer approach cannot effectively address the issue. Our previous results in § 5.2.1 have exhibited the benefits of these new techniques.

5.4 Sensitivity Study

Buffer sizes: Figure 15 shows the impact of the buffer size on request latency. The buffer sizes are varied from 16 MB to 256 MB. We only show the average request latency and do not plot latency CDF as they help make the similar conclusion. We have two observations. First, for both Baseline and PC-LRU, the average latencies decrease as the buffer size increases for all workloads. Second, PC-LRU exhibits superior I/O performance than Baseline for all buffer sizes and workloads: the average latencies with PC-LRU are reduced by up to 24.7%.

Chip latencies: Figure 16 shows the impact of the chip latency on request latency. We take an SLC chip and a TLC chip as an example. For SLC, we set tRead=20us, tProgram=200us, and tErase=1500us. For TLC, these parameters are 75us, 1200us, and 4500us, respectively. As shown, WAFLASH exhibits superior performance over Baseline. For example, as the chip latency increases, the request latency is reduced by 32.6% to 38.5%. This result shows that WAFLASH can efficiently accommodate flash techniques with high storage capacity and long access latency.

5.5 Benchmark-Driven Evaluations

In this subsection, we evaluate WAFLASH on the emulation platform with Filebench [10]. The flash page size is set to 16 KB. For the random write workload, there are 3 threads. For Fileserver, the number of files is 10000, the number of threads is 50, and the mean file size is 128 KB. For Webserver, the number of files is 1000, the number of threads is 100, and the mean file size is 16 KB. For Webproxy, the number of files is 10000, the mean file size is 128 KB, and there are 100 threads.

Figure 17 shows the I/O throughput across 4 Filebench personalities. The results include kernel, file-system, and QEMU overhead in addition to device-level latencies. As Filebench only reports throughput, we do not plot average latency and latency CDF in the results. With WAFLASH, the I/O throughput of the baseline system is increased by up to 109.6%, 12.9%, 9.9%, and 73.8%, respectively. Among the three macro workloads, Webproxy achieves the best improvement because it has the highest unaligned write ratio due to more file accesses and meta-data operations.

5.6 Overhead Analysis

PC-LRU overhead: For PC-LRU, the time complexity of page staging and destaging is the same as the regular LRU based replacement algorithm. The space overhead is mainly caused by tracking PCDs. If we assume that the page size is 4 KB and write buffer size is 256 MB, the space overhead of PCDs is ∼0.1%. The space overhead will be significantly decreased with the page size being increased when SSDs become larger in capacity.

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 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 § 3.2, we control the additional meta-data overhead to be acceptable by partition existing mapping entries in FTL with new fields and using direct and indirect indexing techniques to reduce the overhead.

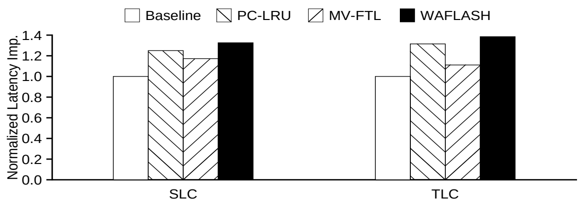


Figure 16: **Chip configuration impact on performance.**

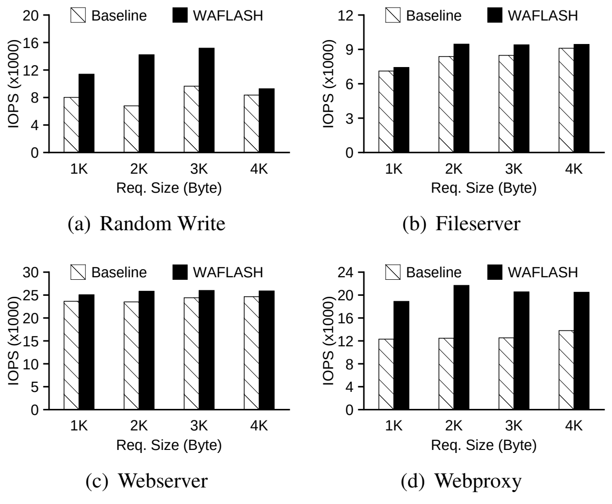


Figure 17: **Filebench on the emulation-based platform.**

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 4 KB and the SSD size is 256 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 ∼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 increase in real workloads.

6 Discussion

While our approach is very promising, it has its limitations. First, when a RAM buffer is used, the integrity of the file system may be compromised for sudden power failures. However, this issue is not caused by our method and it widely exists in other RAM-based buffer systems as well [29]. A possible solution is to use a small battery or capacitor to delay the shutdown until the RAM content is saved to flash memory [61]. We can also use non-volatile memory (e.g., PCM) instead of volatile RAM as the buffer.

Second, operation system (OS) page cache or buffer may compromise the efficiency of WAFLASH because they help to issue page-aligned requests to devices. However, not all applications use OS cache to access devices. Furthermore, most existing applications and OS use block interface to access SSDs without knowing the change of flash page sizes. When the flash page size is larger than the OS cache page size as the density of flash memory increases [31, 55], SSDs will receive a large number of unaligned writes and WAFLASH will benefit them.

Third, we currently only implemented WAFLASH in page-level buffer schemes and page-level FTLs, without testing its behaviors in other types of firmware schemes (see § 7). We choose 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 [14]. 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 [36]. We leave these as our future work.

7 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 [7, 13, 26, 42, 23]. 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 [52], CFLRU [48], and FOR [40]), block-level schemes [25, 29, 51, 28, 8, 18, 47], and hybrid schemes [59, 57].

Among these designs, MCA [52] is the only partial page-aware scheme that is mostly related to WAFLASH. We have discussed the limitation of MCA in § 5.3. 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, MCA delays the eviction of a partial page for a relatively short time while WAFLASH keeps partial pages in the buffer for a longer time by utilizing ACD, so that more full pages can be generated. Furthermore, WAFLASH goes further in the FTL layer when the efficiency of write buffer is limited for serving unaligned writes.

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 [6, 14, 5, 41, 64, 37], *block-level* schemes, and *hybrid* schemes [30, 36, 27, 35, 58].

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.

File system techniques: Numerous efforts enhance flash access performance by designing specific file systems [44, 39, 38, 32] or optimizing data layout in existing file systems [17]. Other approaches propose specific buffer cache architecture with new storage media [33] and optimize traditional buffer write policy for non-cached page updates [3]. While these approaches are effective, they lack portability for general SSDs. The log-based write techniques [50, 45] share a similar idea of the multi-version data store used by WAFLASH but with differences. The log techniques only use the *last version* to retrieve valid data for *improved access sequentiality* while our approach uses *multiple versions* to retrieve valid data for *hiding additional read delays*. In addition, the log technique applies to all writes while our work only applies to unaligned writes.

Other approaches: Other related optimization methods include I/O scheduling techniques within an SSD [61, 9, 60] and with a hybrid storage system [53, 49]. Some efforts also exist in reducing *GC* overhead with new media-based buffer management [34] and specific multi-write coding methods [22]. Also some researchers have studied the buffer management across multiple SSDs [12] and addressed unaligned accesses on multiple HDDs [63]. Different from the above schemes, our work specifically addresses the side-effects of unaligned writes within a single SSD. WAFLASH has better portability because it does not require any modification of existing applications and system software.

8 Conclusions

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 a spectrum of 10 workloads with diverse I/O characteristics, we show that WAFLASH provides as much as 39.5% and 109.6% improvement in I/O latency and throughput, respectively.

References

[1] A. Huffman. NVM Express 1.1a Specifications. http://www.nvmexpress.org, Sep 2013.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 57–70, 2008.

[3] D. Campello, H. Lopez, L. Useche, R. Koller, and R. Rangaswami. Non-blocking Writes to Files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 151–165, 2015.

[4] A. Caulfield, L. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–228, 2009.

[5] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proccedings of the 9th conference on File and storage technologies (FAST)*, pages 77–90, 2011.

[6] M. Chiang, P. C. Lee, and R. Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software: Practice and Experience (SPE)*, 29(3):267–290, 1999.

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

[8] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja. Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks. In *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9. IEEE, 2009.

[9] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung. Exploiting Intra-Request Slack to Improve SSD Performance. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 375–388, 2017.

[10] Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page.

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

[12] 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)*, 66:1890–1902, 2017.

[13] B. S. Gill and D. S. Modha. WOW: Wise Ordering for Writes-Combining Spatial and Temporal Locality in Non-Volatile Caches. In *Proceedings of the 4th USENIX Conference of File and Storage Technologies (FAST)*, 2005.

[14] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, 2009.

[15] 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 *Proceedings of the14th USENIX Conference on File and Storage Technologies (FAST)*, pages 263–276, 2016.

[16] 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.

[17] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 119–133, 2015.

[18] J. Hu, H. Jiang, L. Tian, and L. Xu. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. In *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 69–78, 2010.

[19] 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 *Proceedings of the International Conference on Supercomputing (ICS)*, pages 96–107, 2011.

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

[21] SNIA IOTTA: Storage Networking Industry Associations Input/Output Traces, Tools, and Analysis. `http://iotta.snia.org`.

[22] A. Jagmohan, M. Franceschini, and L. Lastras. Write Amplification Reduction in NAND Flash through Multi-Write Coding. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[23] 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 *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST)*, volume 4, pages 101–114, 2005.

[24] S. Jiang and X. Zhang. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 31–42, 2002.

[25] 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)*, 52(2):485–493, 2006.

[26] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

[27] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th International Conference on Embedded Software (ICES)*, pages 161–170, 2006.

[28] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Transactions on Computers (TC)*, 58(6):744–758, 2009.

[29] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2008.

[30] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A Space-Efficient Flash Translation Layer for Compact-Flash Systems. *IEEE Transactions on Consumer Electronics (TCE)*, 48(2):366–375, 2002.

[31] J. Kim, S. Park, H. Seo, K. Song, S. Yoon, and E. Chung. Nand flash memory with multiple page sizes for high-performance storage devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):764–768, Feb 2016.

[32] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[33] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-Volatile Memory. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, pages 73–80, 2013.

[34] E. Lee, J. Kim, H. Bahn, and S. Noh. Reducing Write Amplification of Flash Storage through Cooperative Data Management with NVM. In *Proceedings of the 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, 2016.

[35] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *SIGOPS Operating System Review (OSR)*, 42(6):36–42, 2008.

[36] 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)*, 6(3), 2007.

[37] C. Li, D. Feng, Y. Hua, F. Wang, C. Jiang, and W. Zhou. A Log-aware Synergized Scheme for Page-level FTL Design. In *Proceedings of the 2017 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1080–1085, 2017.

[38] Y. Lu, J. Shu, and W. Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 75–88, 2014.

[39] Y. Lu, J. Shu, and W. Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2013.

[40] Y. Lv, B. Cui, B. He, and X. Chen. Operation-Aware Buffer Management in Flash-Based Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 13–24, 2011.

[41] D. Ma, J. Feng, and G. Li. LazyFTL: A Page-Level Flash Translation Layer Optimized for NAND Flash Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12, 2011.

[42] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference of File and Storage Technologies (FAST)*, pages 115–130, 2003.

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

[44] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, page 139154, 2012.

[45] B. Nguyen, H. Tan, and X. Zhang. Large-Scale Adaptive Mesh Simulations Through Non-Volatile Byte-Addressable Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[46] Open NAND Flash Interface Specification 3.1. `http://filebench.sourceforge.net/wiki/index.php/Main_Page`.

[47] S. K. Park, Y. Park, G. Shim, and K. H. Park. CAVE: Channel-Aware Buffer Management Scheme for Solid State Disk. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pages 346–353, 2011.

[48] 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)*, pages 234–241, 2006.

[49] J. Ren and Q. Yang. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 278–289, 2010.

[50] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, 1991.

[51] D. Seo and D. Shin. Recently-Evicted-First Buffer Replacement Policy for Flash Storage Devices. *IEEE Transactions on Consumer Electronics (TCE)*, 54(3), 2008.

[52] J. Seol, H. Shim, J. Kim, and S. Maeng. A Buffer Replacement Algorithm Exploiting Multi-Chip Parallelism in Solid State Disks. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–146, 2009.

[53] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *Proccedings of the 8th conference on File and storage technologies (FAST)*, pages 101–114, 2010.

[54] 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.

[55] C. Sun, A. Soga, T. Onagi, K. Johguchi, and K. Takeuchi. A workload-aware-design of 3d-nand flash memory for enterprise ssds. In *Fifteenth International Symposium on Quality Electronic Design*, pages 554–561, March 2014.

[56] 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`.

[57] Q. Wei, C. Chen, and J. Yang. Cbm: A cooperative buffer management for ssd. In *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2014.

[58] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada. WAFTL: A Workload Adaptive Flash Translation Layer with Data Partition. In *Proceedings of the IEEE 27th Symposium onMass Storage Systems and Technologies (MSST)*, pages 1–12, 2011.

[59] G. Wu, B. Eckart, and X. He. BPAC: An Adaptive Write Buffer Management Scheme for Flash-Based Solid State Drives. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2010.

[60] G. Wu and X. He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST)*, pages 117–123, 2012.

[61] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference of File and Storage Technologies (FAST)*, pages 15–28, 2017.

[62] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha. VSSIM: Virtual machine based SSD simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2013.

[63] X. Zhang, K. Liu, K. Davis, and S. Jiang. iBridge: Improving Unaligned Parallel File Access with Solid-State Drives. In *Proceedings of of the 28th IEEE International Parallel and Distributed Processing Symposim (IPDPS)*, pages 381–392, 2013.

[64] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou. An Efficient Page-level FTL to Optimize Address Translation in Flash Memory. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, pages 1–16, 1999.