Re-aligning across-page requests for flash-based solid-state drives

Zhigang Cai, Chengyong Tang, Minjun Li, François Trahay, Jun Li, Zhibing Sha, Jiaojiao Wu, Fan Yang, Jianwei Liao

Re-aligning Across-page Requests for Flash-based Solid-state Drives

Zhigang Cai
Southwest University of China
China

Chengyong Tang
Southwest University of China
China

Minjun Li
Southwest University of China
China

François Trahay
Telecom SudParis
France

Jun Li
Southwest University of China
China

Zhibing Sha
Southwest University of China
China

Jiaojiao Wu
Southwest University of China
China

Fan Yang
Southwest University of China
China

Jianwei Liao*
Southwest University of China
China

Abstract

In flash-based solid-state drives (SSDs), certain small unaligned I/O requests span two logical pages though their size is not larger than the basic write/read unit of SSDs (i.e. an SSD page), and we term them as **across-page** requests. Servicing such across-page requests triggers two separated I/O operations on different SSD pages, and thus impacts the I/O performance and the endurance of SSDs. For mitigating negative effects caused by across-page requests, this paper proposes a novel flash translation layer (FTL) scheme for SSDs to separately re-align such requests via remapping them onto a single SSD page. Consequently, both read and write requests on the across-page data can be completed with one page-level I/O operation. Through a series of experiments based on the selected disk traces of real-world applications, we demonstrate that the proposed realigning method at FTL of SSD devices, can noticeably reduce the I/O latency by between 4.6% and 11.6%, and the erase number (i.e. the indicator of SSD endurance) by between 6.4% and 19.11%, compared to state-of-the-art methods.

CCS Concepts: • **Computer systems organization → Embedded software**.

Keywords: Solid-state Drives (SSDs), Unaligned I/Os, Across-page Remapping, I/O Performance

1 Introduction

NAND flash-based solid-state drives (SSDs) are widely leveraged as external storage because of the advantages of attractive I/O performance and lower energy consumption [1, 2]. The internal structure of an SSD exhibits a hierarchical architecture, and follows the organization of channel-chip-die-plane-block-page from the top downwards [3].

Since the SSD device does not allow in-place update [4], its special firmware, called flash translation layer (FTL) maintains a mapping table to keep track of the current location of a data page, that is referred to as a logical page number (LPN) corresponding with a specific physical page number (PPN) of underlying flash cell [5, 6]. Specially, FTL is responsible for parsing a macro request that may span several pages into several page-sized transactions, and mapping them with multiple PPNs associating with SSD pages.

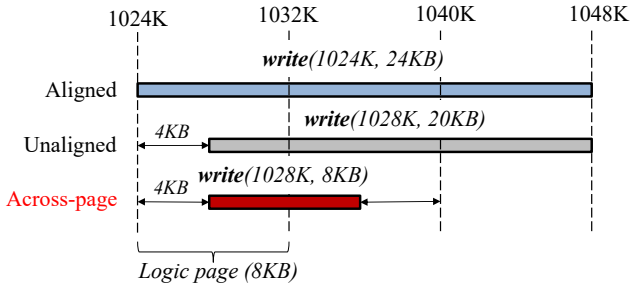


Figure 1. Aligned, unaligned and across-page I/O requests in SSDs (page size: 8KB).

On the other side, the SSD storage device services not only aligned I/Os, but also unaligned I/Os from small I/O access of application [1, 7], as shown in Figure 1. The issue is that the unaligned I/Os may cause extra read and write operation and thus lead to a degradation in I/O performance and endurance in SSDs [8, 9]. This is because an unaligned I/O request must be transformed into aligned I/O requests involving with multiple SSD pages, even though its size is not larger than an SSD page [10].

Since the basic unit of read/write operation of SSDs is the SSD page (typically 4 to 16KB) [11], and the aligned I/Os issued from OS are generally in the 4KB granularity (e.g. VFS in Linux), this mismatch problem between OS and SSDs may result in some aligned I/Os being translated as unaligned I/Os on SSDs (e.g. ***write****(1028K, 8K)* in Figure 1). Moreover, in some scenarios like Virtual Desktop Infrastructure (VDI), Virtual Machines (VMs) are built on the top of file system of host machine. Though the file data on VMs are organized as aligned blocks, the block boundaries might lose when translating to the disk image file on the host machine. Briefly, the aligned I/Os of application running on VMs are likely

*Corresponding author, e-mail: liaotoad@gmail.com.

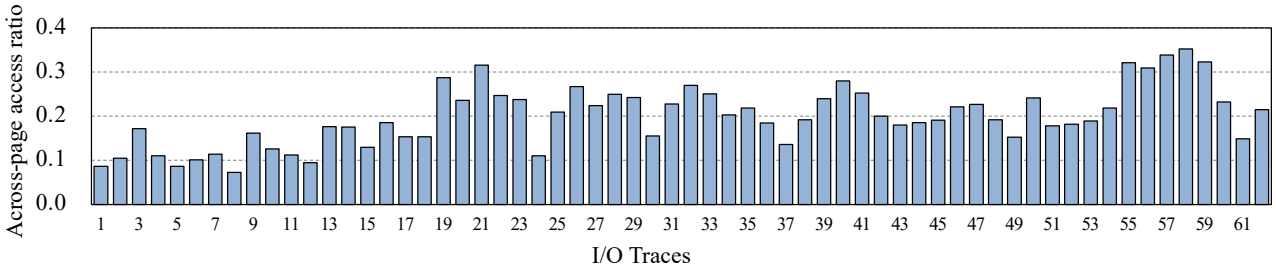


Figure 2. The ratio of across-page access after replaying the traces from the LUN block collection [12]. The number in the X-axis is the sequential number of the trace in the collection folder of *systor17-additional-01*.

to be translated into unaligned ones to the SSDs of host machine, thus worsen the unaligned problem.

More importantly, we define a special case of unaligned request, called **across-page** request, while the request size is smaller or equal to an SSD page (see the last case in Figure 1) [7]. In other words, an across-page request spans two logical pages and will be fulfilled with two separated I/O operations on the underlying flash array of SSD, even though the request size is not larger than an SSD page.

To verify across-page accesses are common in VDI-like applications, We replayed block I/O traces that are recently collected from a part of an enterprise VDI [12], and Figure 2 shows their results[1] of across-page requests ratio to total requests with the page configuration of 8KB. As shown, a significant portion of requests are across-page accesses, and thus we assert that the across-page access feature is not uncommon in real-world applications, such as those run in VDI environments.

In order to minimize the negative effects of unaligned I/Os, several existing works merge the small size data into a relatively large unit to reduce the number of unaligned accesses. Fareed et al. [13] proposed to merge the appropriate sectors with similar update frequency, for alleviating the write amplification problem caused by frequent updates to some sectors of a page. Chen et al. [14] offered a multiregional space management (*MRSM*) design to support subpage-level management while adaptively adjusting mapping granularity by considering the application behaviors. However, *MRSM* introduces a complicated mapping data structure and brings about space and time overhead of address translation.

To address the issue of efficiently processing across-page I/O requests in SSDs, we propose a novel FTL scheme, called *Across-FTL*. It can noticeably reduce the number of extra read/write operations caused by small unaligned I/Os and then offer better performance and longer lifetime of SSDs. In brief, this paper makes the following three contributions.

- We propose to remap across-page write requests onto a specific SSD page, with a two-level mapping table at FTL. Both write and read requests on the across-page data can be satisfied through accessing the single SSD page, so that I/O performance and life time of SSDs can be noticeably enhanced.
- We introduce a merging and rollback scheme to service the update requests that overlap with the remapped across-page data. It merges the updated data with the across-page data while the merged data is not larger than an SSD page, and then remaps all relevant data to another SSD pages.
- We carry out a series of simulation evaluation by replaying 6 block traces of real world applications. As our measurements indicate, our proposal of *Across-FTL* effectively increases I/O performance by an average of 8.4% and reduces the number of erase operations by between 6.4% and 19.11%, in contrast to the-state-of-art methods.

The remainder of this paper is organized as follows: Section 2 presents the background and motivation of our proposed *Across-FTL* scheme. The specifications on the design and implementation of *Across-FTL* are described in Section 3. Section 4 depicts the evaluation methodology and discusses the results. At last, the paper is concluded in Section 5.

2 Background and Motivations

2.1 Overview on I/O Workflow in SSDs

The NAND flash-based SSD consists of several major components. The flash array is used to store data, the I/O interface is used to communicate with the applications running on the host computer, and the SSD controller. The main software layer of flash translation (FTL) runs on the controller, and manages the flash array [15].

Because the basic I/O unit of SSDs is the SSD page, a read/write request may be divided into a number of page-level read/write operations, called as sub-requests [16, 17]. Note that it is regarded the current request has been completed, *if-and-only-if* all associated sub-requests are finished. To fulfill an I/O request, FTL first maps the logical address of read/write operation to the physical address on the flash

[1]The *LUN* trace collection consists of 438 pieces of traces and organizes them as 7 folders. We present the results of all traces in the first folder of *systor17-additional-01*.

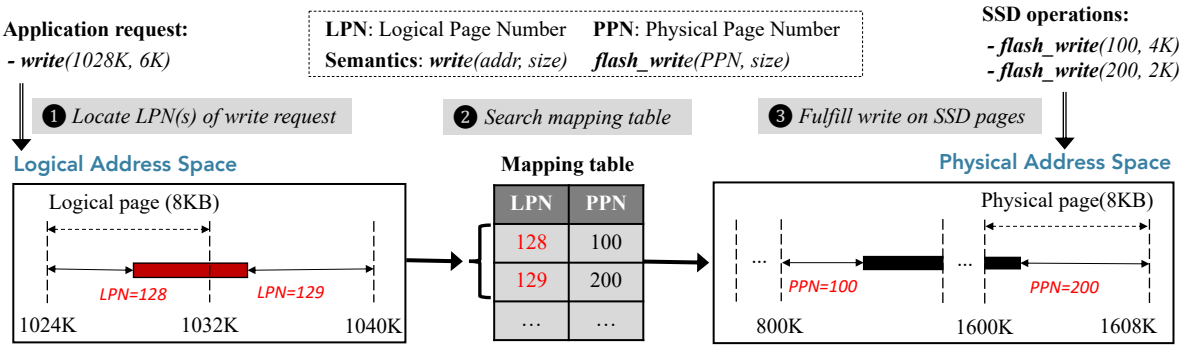


Figure 3. The process of serving an across-page write request in conventional SSDs (page size: 8KB)

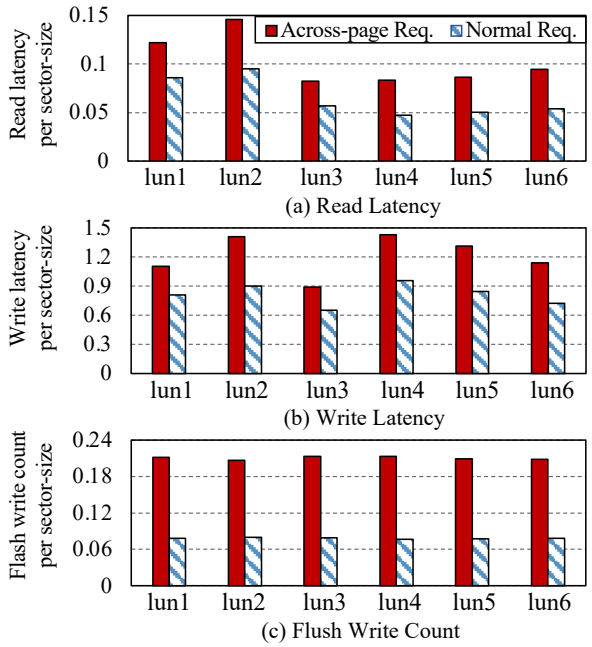


Figure 4. Results of average read latency (a), average write latency (b), and average flush write count (c), per sector-size of across-page requests and other requests, after replaying the selected traces.

array by resorting to the mapping table. After that, the request will be ended after obtaining/flushing the relevant data from/to the flash array with a page granularity [14].

When the available space of SSD becomes low, a time-consuming process, i.e. garbage collection (GC) is executed to reclaim the space occupied by invalided data pages [18]. In the GC process, valid data pages from the GC block will be moved to another new data block, then the outdated data block can be erased and renewed.

2.2 I/O Optimization on Unaligned Request

Unaligned I/Os from small I/O access of application may cause extra I/O operations and then lead to a degradation of I/O performance in SSDs [19]. To minimize the side effects of unaligned access, some existing works merge small size data into to a relatively large block, since the number of unaligned access is greatly reduced [20]. Fareed et al. [13] proposed a scheme that merges sectors with similar update frequency to cut down the overhead caused by frequent updates to some sectors of a page. More importantly, Chen et al. [14] proposed the *MRSM* scheme to adaptively adjust the mapping granularity at the subpage level, by taking user behaviors into account, while facilitating the GC efficiency via the address mapping information. It is true that *MRSM* overcomes the problem of unaligned access, but it makes use of a complicated mapping data structure to record the offset and size information of different piece of data.

The across-page request is a special case of unaligned access, which has no more than one SSD page of data. Figure 3 demonstrates the process of dealing with an across-page write request in conventional SSDs. As seen, it first transforms the logical address of request (i.e. ***write**(1028K, 6K)* in the figure) to the target LPNs of `128` and `129`. Then, FTL maps the LPNs to the corresponding PPNs of `100` and `200` on the flash array by resorting to the mapping table. At last, the write data is flushed onto the corresponding SSD pages.

2.3 Motivations

We suggest that the across-page request causes accesses on two SSD pages, with two major issues impacting SSD performance: **(1)** the page lifetime, which is the time interval between two consecutive writes to the same page must decrease, since pages are updated more frequently than in the original workload. **(2)** the access distribution of the workload is changed, because the accesses to different pages, with different original access frequencies, are now considered as accesses to the same larger page.

In order to quantify the degradation of I/O performance and lifetime caused by dealing with across-page I/O requests in SSDs, we divide the I/O requests into two categories, across-page requests and normal requests, and then recorded the average latency and the induced erase operations after replaying the selected block traces. See Section 4.1 for the details on the benchmarks and the experimental platform. Figure 4 presents the results of average latency and the number

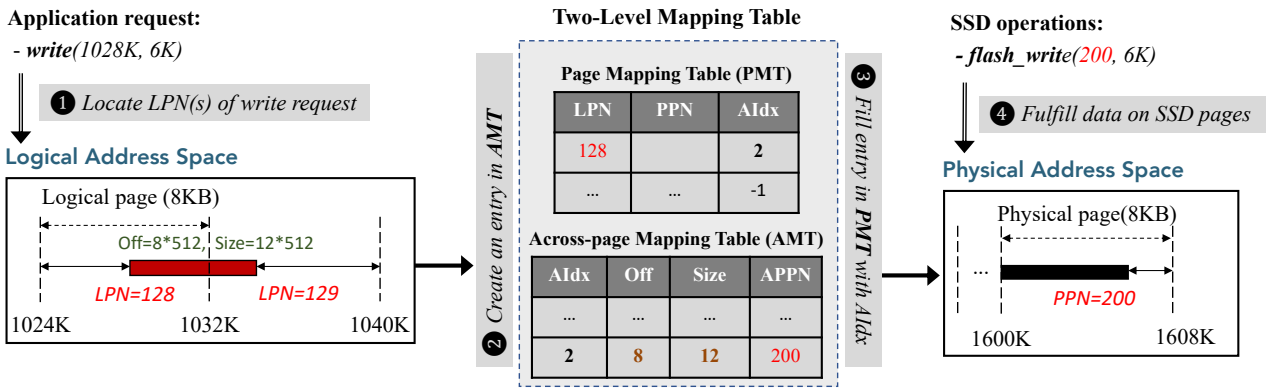


Figure 5. High-level overview of the proposed *Across-FTL* Scheme (page size: 8KB).

of flush operations per sector-size (i.e. 512B) of across-page requests and normal aligned I/O requests.

As seen in Figures 4(a) and 4(b), the read latency and write latency of across-page requests are 1.61 and 1.49 times that of normal I/O requests on average. Figure 4(c) shows the number of flush operations per sector-size of across-page requests and normal I/O requests. As seen, the flush count of across-page request is 2.69 times that of normal I/O request on average, verifying that across-page requests cause more I/O time, as well as more flush operations.

Such observations motivate us to deal with across-page I/O requests that are not larger than an SSD page but span more than one SSD pages. Thus, this paper presents a novel FTL scheme, which remaps the target logical page of across-page requests onto a single SSD page though it involves two logical pages. As a result, it can decrease the number of extra read/write operations, as well as extend the life-time of SSD.

3 Design of *Across-FTL*

This section presents the design and implementation of the proposed approach of FTL scheme (called *Across-FTL*), to fulfill across-page requests. Firstly, an overview of the proposed mechanism is described. Then, it depicts a two-level mapping table in *Across-FTL*, and which is used for mapping the logic address of across-page request to the unique physical page number. After that, both read and write routines with the supports of *Across-FTL* are discussed.

3.1 Architectural Overview of *Across-FTL*

We propose *Across-FTL* to specially allocate an across-page area (i.e. a single SSD page) with another level of mapping table, for holding the data of across-page write request. Then, the across read request can be also fulfilled by remapping it onto the across-page area.

Figure 5 shows an example workflow of servicing an across-page write request to demonstrate the architectural overview of *Across-FTL*, one additional flash write operation is avoided as a result. As seen, *Across-FTL* introduces a new level of mapping table, called across-page mapping table (*AMT*), to record the mapping relationship between PPN and the logical address of across-page data. Then, it marks the index field of *AIdx* in the original page mapping table (*PMT*), to indicate the relevant data pages are remapped to a specific across-page area, and the entry can be retrieved from *AMT*.

As a consequence, in the process of address translation, we first search *PMT* to check whether the logical addresses are remapped in AMT or not. If the logical address is remapped, the translation process is ended by retrieving *AMT*, and the required I/O operation will be finished by operating on the across-page area.

3.2 Two-Level Mapping Table

The two-level mapping table is the critical data structure for address translation in *Across-FTL*, which consists of a page mapping table (i.e. *PMT*) for normal data mapping and an across-page mapping table (i.e. *AMT*) for across-page data mapping. As shown in Figure 5, we add a new index field of *AIdx* in the original page-level mapping table, to indicate whether across-page data has been remapped in the across-page area or not. More exactly, we use "-1" to represent the data is not remapped. The number not smaller than "0" indicates that the data is remapped, and its value represents the index location of relevant mapping entry in *AMT*.

The mapping table of *AMT* is employed to record the mapping information on the remapped across-page data. To be specific, the field of *AIdx* implies the index location of the entry that will be referred in *PMT*; *Off* means the sector offset of the across-page request; *Size* represents the size of the across-page request; the field of across-page physical page number (*APPN*) records the address of the physical page allocated by *Across-FTL* that holds the data from an across-page write request.

3.3 Routines of Across-page I/Os

Because *Across-FTL* focuses on the access on the across-page data, and makes no difference while accessing on other data, this section specifically depicts the routines for different I/O operations on the across-page data in *Across-FTL*.

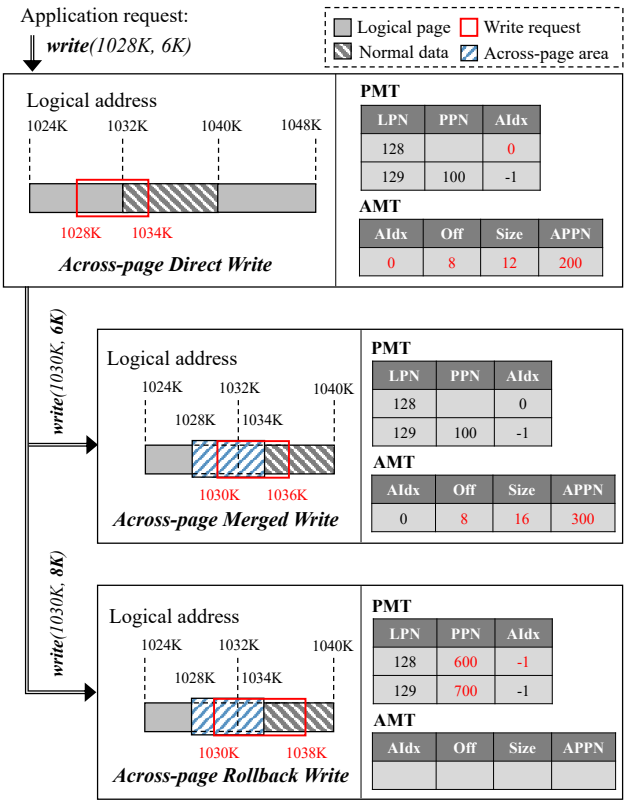
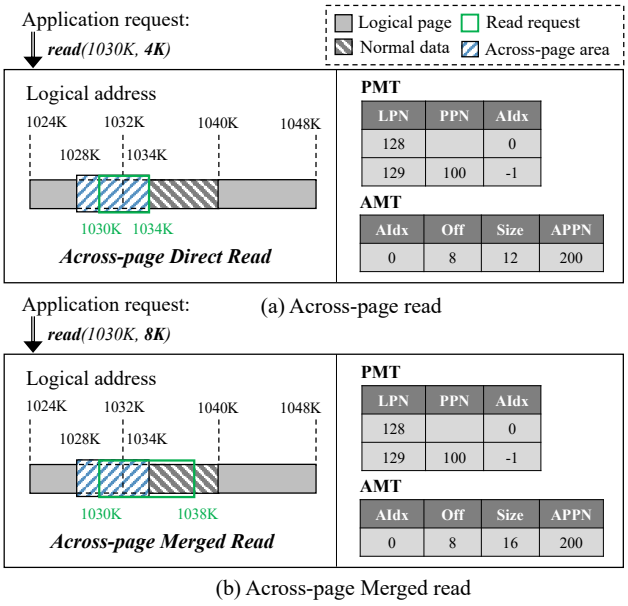


Figure 6. Across-page (merged and rollback) write.



(b) Across-page Merged read

Figure 7. Across-page direct read (a), and merged read (b).

3.3.1 Write Routine. There are two kinds of write requests with respect to the across-page data: the first one is to write the data for the first time (i.e. *write data*) and another one is to update the existing data (i.e. *update data*).

Case of write data: *Across-FTL* checks whether the across-page area of current request has been created or not, by checking the *AIdx* index of related entry in *PMT*. If the index is "−1", it implies this piece of across-page data is the first time to be flushed. Then, *Across-FTL* allocates a free SSD page to store the data of across-page write request as shown in Figure 6, and fills the corresponding entry in the *AMT* table. Next, it sets the field of *AIdx* of the entry in *PMT*, as the index value of newly appended entry in *AMT*. At last, the data will be flushed onto the allocated SSD page.

As seen, an across-page request of *write*(1028K, 6K) has been remapped onto the SSD page whose PPN is 200. Specially, it first creates a relevant entry in *AMT* to log the logical address of request to the across-page area. Then, the index of the entry will be filled in the *AIdx* field of corresponding record in *PMT*, to indicate the original request has been reallocated.

Case of update data: Since the across-page data and original adjacent data are separately saved in different SSD pages, it is necessary to consider the across-page data when satisfying the update requests overlapping with them. To this end, *Across-FTL* supports update policies of across-page merging (i.e. *AMerge*) and across-page rollback (i.e. *ARollback*).

Figure 6 shows servicing an update request spanning from 1030K to 1036K while there is an existing across-page area of (1028K, 1034K). In other words, the total size of the across-page area and the newly appended area is not larger than a page size. In this scenario, the *AMerge* policy can be employed to merge the data in the existing across-page area with the data of current update request, so that the size of merged across-page data is changed from 12 to 16 sectors. Then flush the merged data in a new SSD page according to across-page write manner.

Once the total size of the across-page area and the newly appended area is larger than a page size, it adopts the *ARollback* policy to fulfill the update request. As the example of satisfying *write*(1030K, 8K) shown in Figure 6, it merges the normal data, the existing across-page data and the updated data, then writes merged data in the normal manner. After that, the relevant information about the across-page data in the two-level mapping table will be correspondingly cleared.

3.3.2 Read Routine. Similar to the write routines, there are two scenarios for satisfying a read request related to the across-page data in *Across-FTL*.

Case of direct read: The read request can be serviced by directly accessing the across-page area, while its target data does not exceed the range of the across-page area. Figure 7(a) shows servicing a read request spanning from 1030K to 1034K while there is an across-page area of (1028K, 1034K). In this scenario, *Across-FTL* does bring about an improvement on the read performance, since conventional FTL schemes need to access two SSD pages to obtain the required data,

Table 1. Experimental Settings of *SSDsim* (TLC cell)

Parameters	Values	Parameters	Values
Block number	262144	*Read time*	0.075ms
Page per block	64	*Write time*	2ms
Page size	8KB	*Cache access*	0.001ms
GC threshold	10%	*Cache size*	1‰ capacity

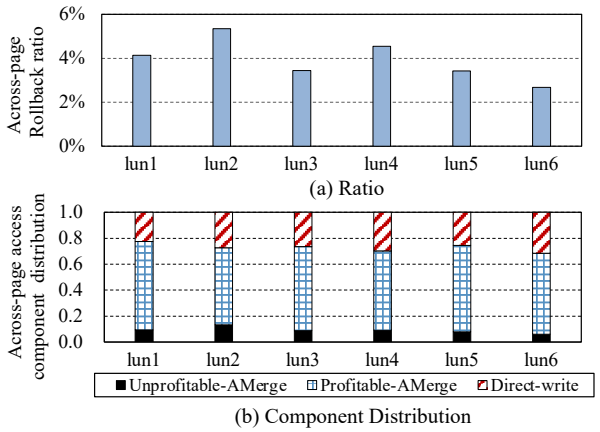


Figure 8. The statistics of across-page access after replaying the selected block traces. (a) Across-page access ratio (b) Across-page access case distribution.

but our proposal does expect reading the data by operating on a single SSD page.

Case of merged read: Once the required data of read request exceeds the range of the across-page area, *Across-FTL* must read the across-page data and the normal data from the underlying flash array, then it merges both kinds of data to respond the read request with the latest version of data. In this scenario, *Across-FTL* cannot yield an improvement on the read performance. As the example presented in Figure 7(b), conventional FTL schemes and *Across-FTL* need accessing two SSD pages for ending the read request.

4 Experimental Evaluation

4.1 Environment Setup

We have performed trace-driven simulation with *SSDsim (ver2.1)*, which has a wide range of configurations and supports of *TLC* flash simulation [17, 21]. We have integrated our proposal with *SSDsim*, for supporting across-page data remapping inside of SSDs. Table 1 demonstrates our settings of *SSDsim* in experiments, which have been also used in prior studies [17, 22]. To reflect the impact of garbage collection, the simulated SSD is aged so that 90% of its capacity has been used [2]. More specifically, valid data occupy 39.8% SSD capacity after warming up through running the trace of *additional-02-2016021710-LUN6* [12].

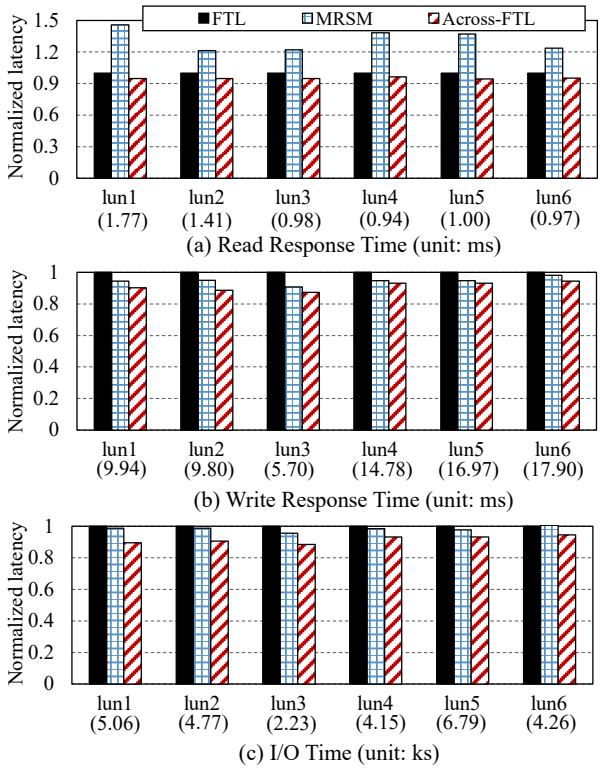


Figure 9. Results of I/O performance metrics of, read latency (a), write latency (b), and overall I/O latency (c). Note that the numbers underlying X-axis are the absolute values of the baseline *FTL*.

Table 2. Specifications on Selected Traces (8KB page size)

Trace	# of Req.	Write R	Write SZ	Across R
lun1	749,806	61.5%	8.9KB	24.7%
lun2	867,967	52.8%	11.3KB	16.4%
lun3	672,580	50.6%	8.6KB	23.4%
lun4	824,068	45.4%	11.2KB	18.7%
lun5	639,558	41.1%	9.2KB	23.5%
lun6	633,234	34.7%	7.6KB	27.5%

Considering the aligned I/Os are commonly translated as unaligned I/Os in virtual machine-relevant applications [23–25]. we employed 6 benchmarks of block traces, that are recently collected from a part of an enterprise VDI [12], to measure the effectiveness of our proposed mechanism. Specifically, we utilized additional-01-*2016021616-LUN1*, *2016021614-LUN0*, *2016021617-LUN2*, *2016021618-LUN6*, *2016021616-LUN4*, and *2016021718-LUN4*, (labeled as *lun1-lun6*) in our tests. The detailed specifications on the selected traces are presented in Table 2. Specially, the metric of **Across R** indicates the ratio of across-page requests to total requests in the traces.

The following three FTL schemes have been implemented in the SSDsim simulator, for comparison evaluation:

- *FTL*, which is the baseline FTL scheme, indicating the default dynamic page-level mapping scheme is enabled, and across-page remapping is not supported.
- *MRSM* [14], which proposes a multiregional space management design to enable subpage-level management while adaptively adjusting mapping granularity by considering the user behaviors. Thus, the issue of across-page access can be removed from the cause, by using a complicated mapping table. Note that, due to the limited capacity of the DRAM buffer, our implementation leaves a major part contents of mapping table in the flash memory, and loads them into the cache if needed.
- *Across-FTL*, which is the proposed method. It supports not only across-page remapping for small I/O requests, but also the update strategies of across-page merge (i.e. *AMerge*) and across-page rollback (i.e. *ARollback*).

4.2 Experimental Results and Discussions

To measure validity of the proposed FTL mechanism that aims to support across-page mapping for small requests, we use the following two major metrics in our tests: (a) *I/O response time* and (b) *erase number*.

4.2.1 Statistics on across-page I/Os in *Across-FTL*. We first collect the statistical data on across-page access while using *Across-FTL*, and Figure 8 shows the results. As seen in Figure 8(a), the ratio of across-page areas performed ARollback to the total across-page areas is `3.9%` on average. In other words, it is verified that most of benefits brought by across-page access are preserved which are the cause of the reduction of I/O latency in *Across-FTL*.

Moreover, we classify the across-page writes into three categories, based on whether such write operations can cause I/O benefits or not, including *Direct-write*, *Profitable-AMerge*, and *Unprofitable-AMerge*. More specifically, *Direct-write* means across-page write presented in Figure 6, which is no existing across-page area involved. We term the *AMerge* operations as *Profitable-AMerge* ones while they are triggered by the across-page requests, as the example presented in Figure 6, and other *AMerge* operations as *Unprofitable-AMerge*. In three kinds of the across-page writes, only *Profitable-AMerge* and *Direct-write* operations bring about I/O benefits, since one additional flash write caused by across-page write request is avoided compared with the conventional FTL scheme. Figure 8(b) presents the distribution of three kinds of across-page writes after replaying the selected traces. On average, we see that only `8.9%` of across-page writes are *Unprofitable-AMerge*, implying a major part of across-page writes can result in I/O improvements with *Across-FTL*.

Considering *merged read*s may lead to additional read operations impacting I/O performance, we also record the number of *merged read* operations. It shows that the flash reads caused by *merged read*s occupy an average of `0.12%` of total read operations, thus they will not noticeably affect read responsiveness.

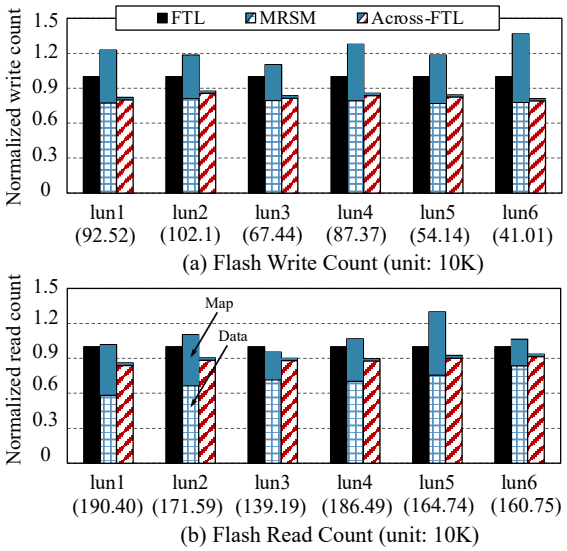


Figure 10. Normalized count of flash write and read operations after replaying selected block traces. In which, the *Map* part indicates the time required for reading/writing the mapping entries, and the *Data* part represents the time needed for reading/writing the user data.

4.2.2 I/O Response Time. I/O response time is the most critical indicator reflecting the performance of SSDs, and Figure 9 reports the normalized results of I/O response time. As illustrated in Figure 9(b), our proposal of *Across-FTL* can cut down the write time by an average of `8.9%` and `3.7%` compared with the baseline of *FTL* and *MRSM* respectively. This is because across-page remapping can reduce the number of flush operations onto the SSD pages while servicing across-page write requests. Consequently, *Across-FTL* can also reduce the time required by read requests by more than `5.0%`, in contrast to other two comparison counterparts.

We see that the most related work of *MRSM* does result in more time for satisfying read requests in the tests, comparing to *FTL* and *Across-FTL*. This is because *MRSM* supports a sub-page level mapping routine to better utilize flash memory, but it needs a very big mapping data structure. Then, a major part of mapping items in the table are held in the flash memory, and only the recently used mapping items are buffered in the DRAM. As a result, it sometimes needs loading the expected part of mapping table into the DRAM cache from the flash memory to service a read request and writes back a part of mapping items to flash memory if the DRAM space is not enough. Similarly, *Across-FTL* holds the additional second level mapping table of *AMT*, it sometimes also requires loading the expected part of the mapping table into the DRAM cache from the flash memory, but with less frequency, in contrast to the most related work of *MRSM*.

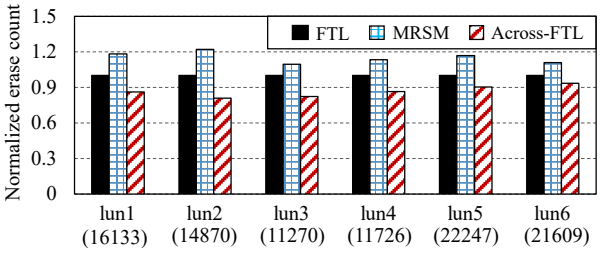


Figure 11. Comparison of erase count after running the selected traces with different FTL schemes.

To further explore the reasons for the decrease in I/O time, we have recorded the count of read and write operations onto the flash memory (i.e. SSD pages) after running all selected block I/O traces. Generally, a larger read/write count implies more access operations on the flash memory, which consequently lead to a longer I/O latency.

Figure 10(a) shows the result of normalized write count after running the selected traces with three FTL comparison methods. Noted that *MRSM* and *Across-FTL* requires to flush not only the write data, but also the contents of mapping table onto the flash memory, so that the write count consists of the normal data writes and the mapping table writes. And the ratio of mapping table writes to total flash writes of *MRSM* and *Across-FTL* are `36.9%` and `2.6%` respectively. As seen, *Across-FTL* can decrease the number of flash writes by an average of `15.9%` and `30.9%`, comparing to *FTL* and *MRSM*. This fact verifies that *Across-FTL* can reduce the count of flush operations on SSD pages, as it requires only one flush operation to fulfill an across-page write request though it originally spans more than one SSD page.

Figure 10(b) shows the result of normalized read count after replaying the block traces. Similar to the results of write count, the read count consists of two parts, normal data reads and mapping table reads. And the mapping table reads of *MRSM* and *Across-FTL* account for `34.4%` and `0.74%` of their total read operations respectively. As shown, *Across-FTL* can reduce the number of flash reads by an average of `9.7%` and `16.1%` comparing to *FTL* and *MRSM*.

Across-FTL can reduce an average of `62.2%` of read operations caused by update requests after re-aligning the across-page requests, in contrast to *FTL*. Specially, *MRSM* supports sub-page mapping, and allows overwriting the old data directly that can greatly reduce the time required for serving the update requests, as reading the old data is not needed in many cases. This is the reason why *MRSM* performs worse than the baseline of *FTL* on the measure of *flash writes*, but yields a better write latency.

Another noticeable clue is that the number of reads and writes on the mapping table of *Across-FTL* is less than that of *MRSM* obviously. This is because *Across-FTL* manages the mapping entries in the unit of SSD page, which requires less memory space (see Section 4.2.4 for details) and less

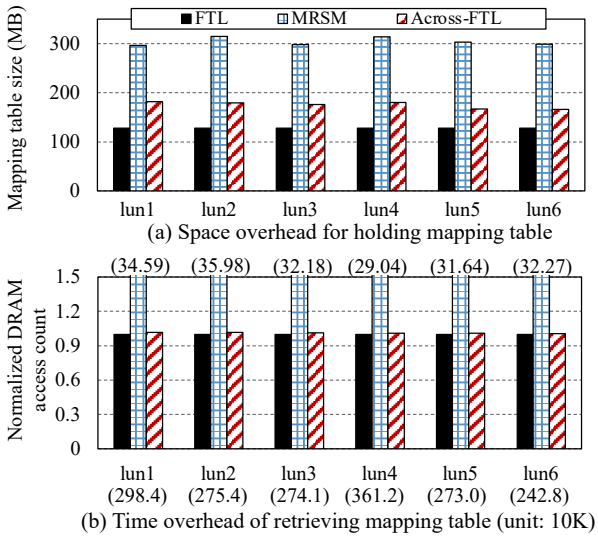


Figure 12. Comparison of space (a) and time (b) overhead after running the selected traces with different FTL schemes

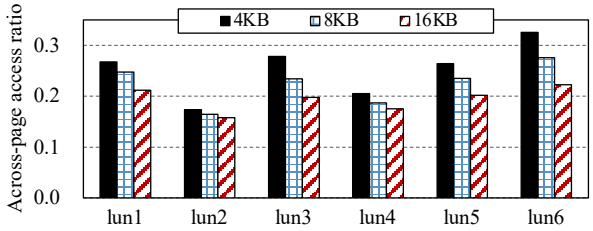


Figure 13. The ratio of across-page accesses with varied sizes of flash page.

operations of mapping entry in/out, comparing to the most related work of *MRSM*.

4.2.3 Erase Number. Once the portion of free SSD space is less than the predefined threshold, garbage collections (GCs) are triggered to reclaim the SSD space by erasing the target SSD blocks. Then, we argue that the reduction of I/O operations on the SSD pages can directly cut down the number of erase operations, which is an indicator reflecting the lifetime of SSD devices since all SSD blocks generally have an erase limit.

Figure 11 presents the results of erase number, and it shows our *Across-FTL* scheme can yield a reduction of erase operations by `13.3%` and `24.6%` respectively, in contrast to *FTL* and *MRSM*. Thus, we conclude that *Across-FTL* can remarkably reduce the number of erase operations and thus optimize SSD life-time.

4.2.4 Overhead Analysis. The main memory overhead of various FTL schemes is due to the storage of the mapping table. For *Across-FTL*, the entries of mapping table are managed in the size of flash page and can be allocated dynamically. Though with an additional mapping table (*AMT*) applied in

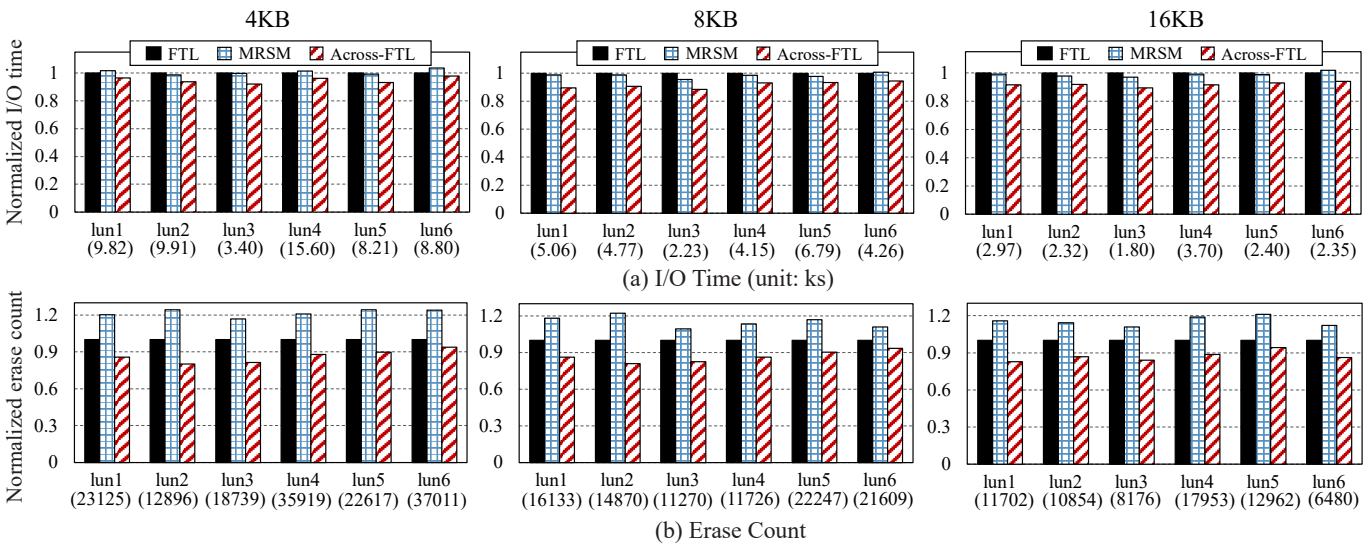


Figure 14. I/O performance and SSD lifetime with varied configurations of page size in SSDs. The results of overall I/O response time (a), and erase count (b).

contrast to *FTL*, as shown in Figure 12(a), the space overhead of *Across-FTL* is 1.4 times that of *FTL* on average.

MRSM requires the largest memory overhead, and the average space overhead is 2.4 times that of *FTL*. This is because *MRSM* is a sub-page level FTL scheme, the number of entries is much larger than that of *FTL* and *Across-FTL*. With the specification shown in Table 1, 42.1% of total mapping entries can be kept in the DRAM cache while using *MRSM*, that worsen the I/O performance of SSDs.

We record the number of DRAM access of three selected FTL schemes and the results are presented in Figure 12(b). Though with a two-level mapping table applied in *Across-FTL*, no more than 1.1% of extra DRAM access on average is created in contrast to *FTL*. This is because *Across-FTL* can lessen DRAM access by reducing the number of write/read operations. For *MRSM*, the number is an average of 32.6 and 32.3 times that of *FTL* and *Across-FTL* respectively. This is because the tree data structure is applied in *MRSM* to store mapping entries and thus more lookup time is expected. The lookup overhead accounts for 0.11% of total I/O time on average after conducting tests on a resource-limited platform that has an ARM Cortex A7 Dual-Core CPU with 800MHz and 128MB of memory.

4.3 Case study

To investigate the performance of our proposal of *Across-FTL* under various configurations of SSD page size, we have carried out a case study to check its effectiveness.

As shown in Figure 13, we first collect the statistical data about the ratio of across-page access after running the benchmarks, under different configurations of SSD page size while using *Across-FTL*. As seen, the number of across-page access keeps decreasing as the size of SSD page becomes larger, this is because a larger flash page can hold more data to refrain from across-page access.

We then measure the performance metrics of the I/O response time and the erase number after replaying the selected I/O traces, with varied size configurations of SSD page with *Across-FTL*. Figures 14(a) and 14(b) show the detailed results of overall I/O time and erases counts respectively. As seen, the proposed *Across-FTL* method outperforms the baseline scheme of *FTL* and *MRSM* under various size configurations of SSD page. This is because *Across-FTL* enables servicing across-page requests by accessing only one SSD page. Consequently, *Across-FTL* can cut down the number of sub-requests that should be submitted to the flash memory and then yield better I/O performance and erase counts that is a metric of SSD lifetime, in contrast to other two comparison counterparts.

We argue that the most important clue shown in Figure 14 is that, the degree of performance improvement and lifetime extension caused by *Across-FTL* does not decrease as the page size increases. This information further verifies *Across-FTL* has good scalability on the size of SSD page, since the performance improvement is only in line with the ratio of across-page requests in the benchmarks, which has been demonstrated in Figure 13.

4.4 Summary

With respect to comparing the proposed scheme to conventionally used FTL schemes for SSDs, we emphasize the following three key observations. *First*, both across-page write and read requests can be serviced by accessing the single SSD page, through purposely remapping them onto a specific SSD page. As a result, I/O performance and life time of SSDs can be enhanced. *Second*, *Across-FTL* does not

noticeably introduce more space overhead and mapping time overhead, and it scales well for varied configurations of page size in SSD products. *Third*, the most related work of *MSRC* introduces the largest number erase counts caused by loading the complicated mapping table into the capacity-limited cache, as it manages the across-page data in a sub-page unit.

5 Conclusion

This paper has proposed and evaluated a novel FTL scheme called *Across-FTL* for SSD devices, which addresses the issue of performance degradation caused by across-page requests. We have introduced a two-level mapping table to support remapping across-page requests to aligned areas of flash array in SSDs, instead of putting them adjacent to the data pages according to the logical address. Furthermore, we have designed both write and read policies that support across-page merging and rollback, to specifically service I/O requests that involve with the across-page data. Through a series of emulation experiments based on several realistic disk traces, we show that the proposed *Across-FTL* scheme can reduce I/O response time by up to 8.4% on average, in contrast to other comparison counterparts.

Acknowledgments

This work is partially supported by the Research Project of Education Department of Hunan Province (No.20C1472), and the Southwest University Postgraduate Scientific Research Innovation Project (No.SWUS23091).

References

[1] Kim B S, Choi J, Min S L. Design tradeoffs for SSD reliability. *USENIX Conference on File and Storage Technologies (**FAST**)*, 2019.

[2] Gao C, Shi L, Di Y, et al. Exploiting chip idleness for minimizing garbage collection–induced chip access conflict on SSDs. *ACM Transactions on Design Automation of Electronic Systems (**TODAES**)*, 2018.

[3] Wang Y, Wang W, Xie T, et al. CR5M: A mirroring-powered channel-RAID5 architecture for an SSD. *Mass Storage Systems and Technologies (**MSST**)*, 2014.

[4] Micheloni, Rino. Solid-state drive (SSD): A nonvolatile storage system. *Proceedings of the IEEE (**PIEEE**)*, 2017.

[5] Kim B S, Yang H S, Min S L. AutoSSD: an Autonomic SSD Architecture. *USENIX Annual Technical Conference (**ATC**)*, 2018.

[6] Elyasi N, Arjomand M, Sivasubramaniam A, et al. Exploiting intra-request slack to improve SSD performance. *International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, 2017.

[7] Shu J, Li F, Li S, et al. Towards Unaligned Writes Optimization in Cloud Storage With High-Performance SSDs. *IEEE Transactions on Parallel and Distributed Systems (**TPDS**)*, 2020, 31(12): 2923-2937.

[8] Kim J, Seo S, Jung D, et al. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Transactions on Computers (**TC**)*, 2011.

[9] Yang J, Li B, Lilja D J. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (**TOMPECS**)*, 2020.

[10] Kim B S, Choi J, Min S L. Design tradeoffs for SSD reliability. *USENIX Conference on File and Storage Technologies (**FAST**)*, 2019.

[11] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. *European Conference on Computer Systems (**Eurosys**)*, 2014.

[12] Lee C, Kumano T, Matsuki T, et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. *ACM International Systems and Storage Conference (**SYSTOR**)*, 2017.

[13] Fareed I, Kang M, Lee W, et al. Leveraging intra-page update diversity for mitigating write amplification in SSDs. *ACM International Conference on Supercomputing (**ICS**)*, 2020.

[14] Chen S H, Tsao C W, Chang Y H. Beyond address mapping: A user-oriented multiregional space management design for 3-D NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, 2020.

[15] Jung M, Kandemir M T. Sprinkler. Sprinkler: Maximizing resource utilization in many-chip solid state disks. *International Symposium on High Performance Computer Architecture (**HPCA**)*, 2014.

[16] Tavakkol A, Mehrvarzy P, Arjomand M, et al. Performance evaluation of dynamic page allocation strategies in SSDs. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (**TOMPECS**)*, 2016.

[17] Zhang W, Cao Q, Jiang H, et al. PA-SSD: A Page-Type Aware TLC SSD for Improved Write/Read Performance and Storage Efficiency. *International Conference on Supercomputing (**ICS**)*, 2018.

[18] Sha Z, Li J, Song L, et al. Low I/O Intensity-aware Partial GC Scheduling to Reduce Long-tail Latency in SSDs. *ACM Transactions on Architecture and Code Optimization (**TACO**)*, 2021.

[19] Yadgar G, Gabel M, Jaffer S, and Schroeder B. SSD-based workload characteristics and their performance implications. *ACM Transactions on Storage (**TOS**)*, 2021.

[20] Kakaraparthy A, Patel J M, Park K, et al. Optimizing databases by learning hidden parameters of solid state drives. *Proceedings of the VLDB Endowment*, 2019.

[21] Xu X, Cai Z, Liao J, et al. Frequent Access Pattern-based Prefetching Inside of Solid-State Drives. *Design, Automation & Test in Europe Conference & Exhibition (**DATE**)*, 2020.

[22] Li J, Sha Z, Cai Z, et al. Patch-based data management for dual-copy buffers in RAID-enabled SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, 2020.

[23] Kim H, Lim H, Jeong J, et al. Task-aware virtual machine scheduling for I/O performance. *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (**VEE**)*, 2009.

[24] Garraghan P, Ouyang X, Yang R, et al. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing (**TSC**)*, 2016.

[25] Li J, Wang Q, Lee P P C, et al. An in-depth comparative analysis of cloud block storage workloads: Findings and implications. *ACM Transactions on Storage (**TOS**)*, 2023.