PAPA: Partial Page-aware Page Allocation in TLC Flash SSD for Performance Enhancement

Imran Fareed, Mincheol Kang, Wonyoung Lee, and Soontae Kim
Embedded Computing Laboratory
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
Email: {imranf45, mincheolkang, wy_lee, kims}@kaist.ac.kr

***Abstract*—The three bit types, namely, least significant bit (LSB), central significant bit (CSB), and most significant bit (MSB), in the TLC flash memory exhibit variable read/write latencies. Reading/writing an MSB takes more time than reading/writing a CSB, and an LSB incurs the minimum latency. In addition, the increased size of flash pages results in the formation of partial page writes. The partial page writes are significantly costly if they update the existing data, as partial updates perform read-modify-write (RMW) operations for ensuring data integrity. The performance further worsens if the to-be-updated data by partial writes are stored in high-latency MSB pages. Conventional TLC programming designs do not consider the size of the write requests and follow a type-blind page allocation, thereby missing a key opportunity to boost the performance of the TLC flash memory. In this study, we propose a partial page-aware page allocation (PAPA) scheme for TLC flash memory. PAPA simultaneously considers both the write request size and flash page types for performing page allocation. Our study reveals that most of the to-be-updated data updated via partial updates are partial pages. Therefore, the central mechanism of PAPA scheme is to prioritize low-latency LSB pages for partial page writes, as partial page writes incur extra latency to read the existing data, during update operations; however, high-latency CSB/MSB pages are assigned to full page writes. Our analysis using various write-intensive workloads report that the PAPA scheme improves the write response time, RMW latency, and IOPS by 55%, 34%, and 14% on average, respectively.**

***Index Terms*—TLC flash memory, partial updates, variable latencies, performance**

I. Introduction

The triple-level cell (TLC) flash memory has been attracting the attention of the storage market because of its high bit density and low cost per bit compared with its predecessors such as single-level cell (SLC) and multi-level cell (MLC) flash memories. The TLC flash stores three bits, namely, least significant bit (LSB), central significant bit (CSB), and most significant bit (MSB), in a cell. As the aforementioned three bits in a flash cell require different number of memory accesses for reading/writing, they possess different read/write latencies [1]–[3]. Therefore, the three bits are separated to form different types of pages, i.e., LSB, CSB, and MSB pages, each page having variable read/write latencies [2]–[4]. The typical read/write latencies of LSB, CSB, and MSB pages are 50/500, 100/2000, 110/5500 μs, respectively [2], [3].

This work was supported by the National Research Foundation (NRF) grants funded by Korean Government (2018R1A2B2005277).

Owing to the increase in the bit density, to accommodate large volume of data, the size of flash pages has also been increasing, up to 16KB noticed in recent flash memories [5], [6]. However, the write requests from the host system are sent in the units of 4KB sectors which is smaller than the flash I/O unit [7], and thus large number of partial page writes are received by the SSD, either due to the small host request size or due to the misalignment of the sectors [5], [6]. These partial writes degrade both lifetime and performance, which are attributed to the under-utilized space and read-modify-write (RMW) operations that are unavoidably involved in the partial updates [5], [6]. The partial updates further exacerbate the problem if the data to be updated are stored in the high-latency CSB/MSB pages, as partially updating the existing CSB/MSB data requires an additional read to the high-latency CSB/MSB pages that contain the data to be updated. Therefore, it is important to minimize the number of partial updates to long-latency CSB/MSB pages.

To leverage the diverse latencies in the TLC flash memory, several prior works have proposed to assign only LSB pages for all write requests. Grupp et al. [4] suggested to assign LSB pages proactively to the dense write requests. However, the benefit of this work is limited, owing to the strict program order of MLC/TLC blocks. Moreover, Park et al. [8] proposed FlexFTL that uses a relaxed program order to completely exploit the MLC latency asymmetry. They also proved that despite using a relaxed program order, the reliability of the MLC/TLC flash is not compromised. Zhang et al. [3] proposed a page-type aware SSD (PA-SSD), which also used a relaxed program order and assigned same type of pages to all the transactions of a host write request. The motivation of PA-SSD, behind using same page-type for all the transactions of a host write request, is that all the transactions of a write request have same urgency of completion, and therefore these transactions should be assigned the same type of page for achieving high efficiency.

Although the aforementioned techniques can utilize the asymmetric latencies of the MLC/TLC flash memory, they do not consider the RMW overhead during partial updates, while performing page allocation, and therefore they suffer from considerable performance overhead during partial updates. However, our proposed scheme, partial page-aware page allocation (PAPA), considers costly partial updates for page

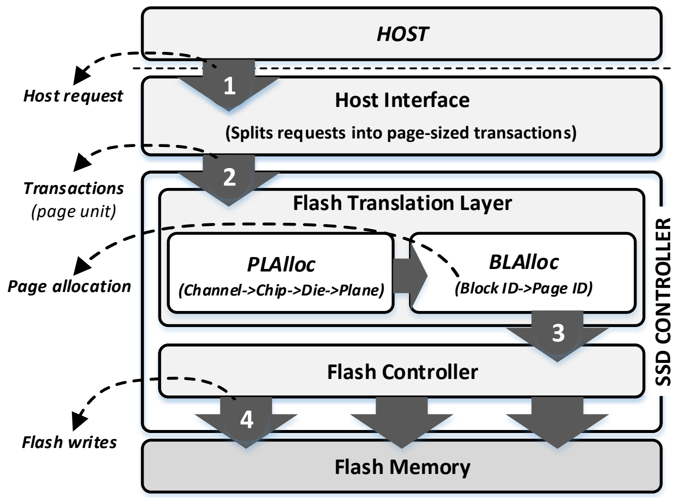

Fig. 1: SSD components and write-request processing [9].

allocation and relies on a relaxed program order to flexibly use the three types of TLC pages. The basic idea of PAPA is to allocate LSB pages to partial writes, as future updates to the stored LSB data require reading the low-latency LSB pages, followed by writing the updated data to the new pages. However, the long-latency CSB/MSB pages are assigned to full page writes, as full page updates do not incur any additional overhead. Because all the partial updates require an extra read operation on the existing data, the response time can be reduced upon storing partial writes in low-latency LSB pages (see Section V).

The remaining sections of this paper are as follows. In Section II, we present the background, and the related works are discussed in Section III. We explain the motivation behind this work in Section IV. In Section V, the details of the proposed scheme, PAPA, are presented. The evaluations and experimental results are explained in Section VI. Finally, we draw conclusions in Section VII.

II. Background

A. SSD Components

The essential components of the modern SSD comprise the *host interface*, *SSD controller*, *flash controller*, and *flash memory*, as depicted in Fig. 1. The *host interface* provides communication between the host and SSD controller. The *SSD controller* receives and processes the host requests with the help of the flash translation layer (FTL), which performs logical to physical mapping. The *flash controller* is the interface between the SSD controller and flash chips. The *flash memory* stores the actual user data received from the host, and it comprises several flash chips. Each flash chip is further composed of few dies, which are further divided into few planes. Several blocks, each comprising many pages, form a plane. The basic I/O operations are performed on page granularity.

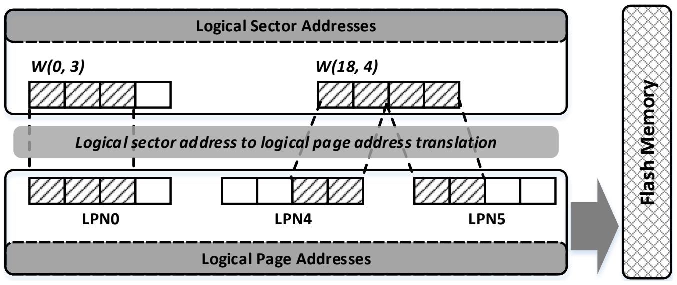

Fig. 2: LSN-to-LPN translation and partial page formation.

B. Address Translation and Partial Writes

*1) **Address translation**:* The host sends write requests that contain logical sector addresses (LSA) (see 1 of Figure 1), which are the basic I/O units of modern file systems. Subsequently, the host interface splits the host write requests into several page-sized transactions, and each transaction is translated to a specific logical page address (LPA) (see 2 of Figure 1) by FTL [9]. Finally, the LPA is translated to physical page address (PPA) with the help of *PLAlloc* and *BLAlloc* primitives for write request completion. The duty of *PLAlloc* is to allocate channel, chip, die, and plane IDs, whereas, the *BLAlloc* is responsible for allocating the block and page IDs, by selecting the free pages for each transaction [3], [9]. Once the address translation is resolved by allocating a specific PPA to each transaction (see 3 of Figure 1), the flash controller writes the user data into the corresponding flash page (see 4 of Figure 1).

*2) **Partial writes**:* Write requests are sent by the host file system in multiple of 4KB sectors [5]–[7]. Many partial writes are generated because of either the small host request size or misalignment of the sectors. Figure 2 depicts the LSA-to-LPA translation in the FTL, and it also illustrates the formation of the partial writes in SSDs in two scenarios. First, upon receiving the write request *W(0, 3)* with sector address 0 and size 3 sectors, the FTL translates it to LPN0. As the request *W(0, 3)* has only 3 sectors, LPN0 is partially filled, and therefore a partial write is generated because of the small size of the write request. The second scenario is when a full page write request *W(18, 4)* is received. Although *W(18, 4)* is a full page request, it is misaligned to the sectors of LPN4 and LPN5 and spans over two pages, thereby generating two partial page writes, which, in turn, degrade the SSD performance, owing to the costly RMW operations involved in the partial updates [6].

C. TLC Flash Page Types and Program Sequence

Unlike its predecessors, such as the SLC/MLC flash, the TLC flash stores three bits per cell, namely, LSB, CSB, and MSB, each exhibiting different read/write latencies [2], [3]. Because the aforementioned bits exhibit variable latencies, those with the same latency within a wordline form a flash page, and the pages within a wordline are programmed in a fixed order, as shown in Figure 3 [3], [4], [11]. The aforementioned variable-latency pages of a wordline are programmed page-by-page according to their given IDs to mitigate the cell-to-cell interference to the programmed wordlines [1], [12].

Owing to the strict program order, the flexibility of writing to a desired page-type is compromised, as only one active block is available for the incoming write requests in the plane allocated by the *PLAlloc* primitive. Once the plane is decided by the *PLAlloc* primitive, the data can only be written to the next assigned page in the current active block allocated by *BLAlloc*. Although the strict program order with only one active block and one candidate page simplifies the meta-data management, it highly limits the flexibility to write to the desired page-type.

III. Related Works

The variable latencies of the various page-types of MLC/TLC flash have been used in different ways in previously conducted studies, to improve the write response time of the SSD. We broadly categorize the existing works as follows.

1) ***SLC flash as write buffer****:* The read/write latencies of the SLC flash memory being lower than those of its successors, provides an opportunity to the researchers to employ the SLC flash as a write buffer in the MLC/TLC SSD for boosting the performance [13], [14]. However, this performance enhancement increases the manufacturing cost of the SSD, owing to the additional SLC chips required for the write buffer. However, the manufacturing cost might be reduced by enabling the SLC mode in the MLC/TLC flash memory [1], [11], [15]–[17]. However, enabling the SLC mode in the MLC/TLC flash memory restricts the storage space available for user data. More importantly, the SLC buffer can become saturated in negligible time for write-dominated user data, owing to the limited size of the SLC flash, thereby degrading the performance of the SSD.

2) ***Utilizing low latencies of LSB pages****:* Owing to their low read/write latencies, LSB pages are the best candidates to enhance the flash I/O performance. Grupp et al. [4] suggested to assign low-latency LSB pages proactively for dense write requests to improve the peak write performance. However, the benefit of this work is limited, owing to the strict program order of MLC/TLC blocks. Moreover, Park et al. [8] proposed FlexFTL that used a relaxed program order to flexibly exploit the MLC latency asymmetry. They also proved

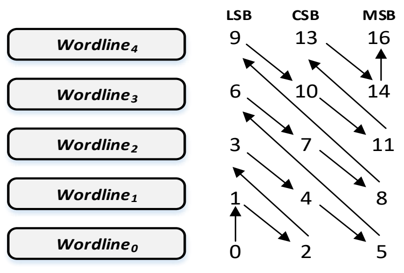

Fig. 3: Strict program order in a conventional TLC block [3], [10], [11].

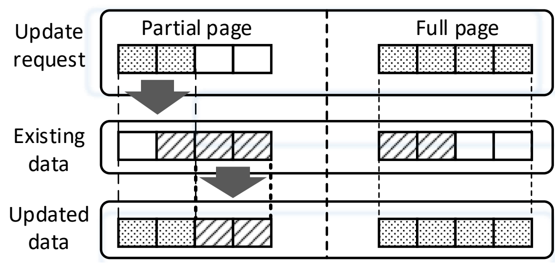

Fig. 4: Difference between partial and full page updates.

that despite using the relaxed program order, the reliability of the MLC/TLC is not compromised, as a strict program order is the over-specification to reduce the inter-cell program interference. Zhang et al. [3] proposed a page-type aware SSD (PA-SSD), which also utilized a relaxed program order and assigned the same type of pages to all the transactions of a host write request. The motivation behind PA-SSD is that all the transactions of a write request have the same urgency for completion, and therefore, these transactions should be assigned the same type of pages for achieving high efficiency. In addition, PA-SSD suggested different scenarios of utilizing LSB pages for enhancing the write performance.

Although the aforementioned schemes can satisfactorily utilize the low write latencies of LSB pages, they do not consider the cost of RMW operations inevitably involved in the partial updates; therefore, the write response time may increase. However, our proposed scheme, PAPA, considers the size of each transaction of a write request, i.e., whether the transaction is partial page or full page, for performing type-directed page allocation, and accordingly allocates LSB pages to partial writes and CSB/MSB pages to full page writes (see Section V).

IV. Motivation

As flash memory does not support in-place update, the existing data should be read, and subsequently modified using the new incoming data, following which the updated data should be written to a new flash page. This operation is known as RMW operation. However, in case of full page updates, RMW operations need not be performed, as they contain the updated version of all the sectors of existing data. Figure 4 depicts the difference between partial and full page updates.

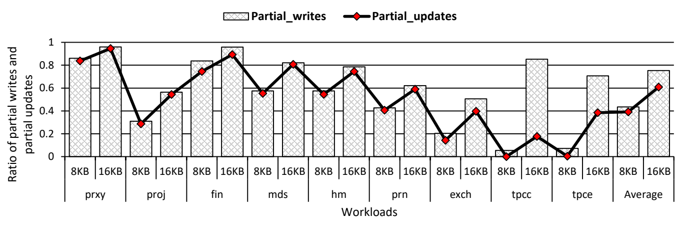

Fig. 5: Ratio of partial writes and partial updates in various workloads for 8KB and 16KB pages. Both the partial writes and partial updates are normalized to the total writes.

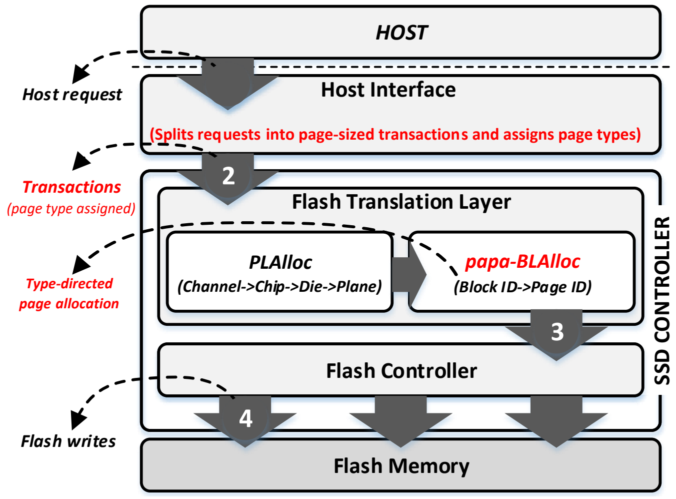

Fig. 6: Write-request processing and page-type assignment. The host interface proactively assigns the page type to each transaction according the size of requested data. The *papa-BLAlloc* primitive which allocates the assigned flash page types to the transactions, is the replacement of the *BLAalloc* primitive in the conventional SSD.

In case of partial updates, the existing data are read into the SSD controller, and the sectors are modified; subsequently, the modified or updated data are written into a new flash page, as shown in left-hand side of Figure 4. However, full page updates involve no overhead of reading the existing data, as the update request contains the updated sectors of existing data, as shown in right-hand side of Figure 4. Thereafter, the updated data are written to the available flash page, and the corresponding obsolete data are invalidated. To conclude, partial page updates are more costly than full page updates, as partial updates perform extra read operations, on the existing data, to avoid data inconsistencies.

In addition, the number of partial page writes outnumber the full page writes in all the workloads, as shown in Figure 5. The figure shows the ratio of the partial writes and partial updates, both normalized to the total number of writes. In an 8KB page, 43% of the write requests are partial write requests and 39% are partial update requests. However, as the page size increases to 16KB, the ratio of partial writes and partial updates increases considerably. From Figure 5, it is evident that 75% of the total write requests are partial write requests, whereas, 61% of the total write requests are partial update requests, in 16KB page. Therefore, considering the high ratio of both partial writes and partial updates in the workloads, we recommend to allocate low-latency LSB pages to the partial writes, to mitigate both the RMW latency and write response time.

Unlike SLC and MLC flash memories, the TLC flash memory stores three bits per cell, each with different read/write latencies. The same type of bits in a wordline constitute a page. The TLC flash exhibits variable read/write latencies because of different number of memory accesses required to perform

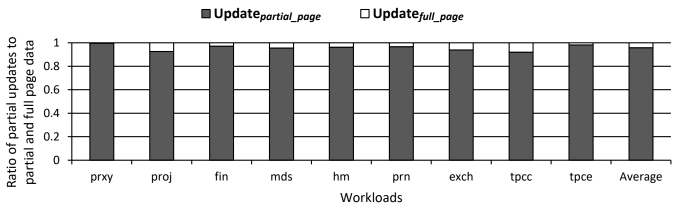

Fig. 7: Ratio of partial updates to partial and full page data.

read/write operations on LSB, CSB, and MSB pages [1]–[3]. The aforementioned three types of pages in the TLC flash are programmed using a strict program order, called shadow programming, to alleviate the cell-to-cell program interference [2], [8], [9]. In addition, Tavakkol et al. [9] proved that a strict program order is an over-provision to mitigate the cell-to-cell program interference, and instead proposed a relaxed program order to leverage the benefits of low-latency pages in MLC/TLC flash. We also use the relaxed program order to flexibly utilize the low-latency LSB pages for storing costly partial writes.

V. PAPA: PARTIAL PAGE-AWARE PAGE ALLOCATION

Considering the costly RMW operations involved in partial updates, to enhance the SSD performance, we propose a partial page-aware page allocation (PAPA) scheme for TLC SSD that proactively allocates LSB pages to partial writes, and writes the full page data to CSB/MSB pages. Figure 6 depicts the write-request processing in PAPA scheme. The write-request processing both in the conventional SSD and proposed PAPA schemes are similar, except that the latter proactively assigns the page-type to the user requests according to the size of each transaction, and then the proposed *papa-BLAlloc* allocates the corresponding flash page, to each transaction, by allocating the block ID and page ID.

A. Type-directed Page Allocation

The LSB pages are preferred for writing partial page data, for reducing the read/write response time and RMW latency which is inevitably involved in the partial updates. Some astute readers may suggest that the incoming partial writes may update the existing full page data, and thus the RMW latency might be high due to reading the high latency CSB/MSB pages (as full page data is written in CSB/MSB pages). In other words, what is the assurance that the partial writes will only update the existing partial pages and not the full pages. To answer this question, we performed extensive experiments with a rich set of workloads to measure the ratio of partial updates updating the existing partial pages and the full pages. Figure 7 presents the aforementioned ratio of partial and full pages updated by the partial updates. As evident from the figure, even in the worst case (in *tpcc* trace), more than 92% of the partial updates target the existing partial pages. In addition, on average, 96% of the partial updates target the existing partial pages. Hence, nearly all the partial update requests target the existing partial pages, and thus we can store the partial pages in the low-latency LSB pages to mitigate both the RMW latency and write response time.

The efficiency of PAPA depends on the availability of the assigned page-types in active blocks, i.e., free LSB pages for partial page writes and free CSB/MSB pages for full page writes. The conventional SSD maintains only one active block for each plane, and therefore, only one candidate page is available for the incoming write requests [10]. Although this approach simplifies the flash resource management, it has no ability for type-directed page allocation. Although one might modify the *PLAlloc* primitive to select a plane whose next candidate page is the desired page-type, it would limit the parallelism within the SSD by selecting a fixed path from the channel to plane [9], [18]. Therefore, we would like to provide more than one candidate pages within each plane to satisfy the requirements for performing the type-directed page allocation in PAPA scheme. Unlike the conventional SSD, to perform the proposed type-directed page allocation, two active blocks per plane are simultaneously required in the PAPA scheme, resembling to more than one write points suggested in [4].

B. Managing Multiple Active Blocks in a Plane

In Section II-C, we discussed that the conventional SSD has a fixed program sequence and that it provides only one active block within a plane, thereby considerably compromising with the flexibility to choose the desired page-type. Therefore, multiple active blocks within each plane is a key requirement in designing the PAPA scheme. We can fulfill the requirement of writing the desired page-type by wisely relaxing the program constraints in each block and providing more than one active block in each plane.

The fixed program order in the conventional SSD was proposed to mitigate the cell-to-cell interference problem, by ensuring that a fully-programmed wordline experiences interference at most from one adjacent page. For example, before page 8 is programmed in *wordline 1*, all the adjacent pages are programmed except page 11 in *wordline 2*, as shown in Figure 3. Therefore, a fully-programmed *wordline 1* experiences the interference only from page 11 in the adjacent wordline.

We can formalize the strict program order in the TLC flash to explore and leverage a more flexible program order that

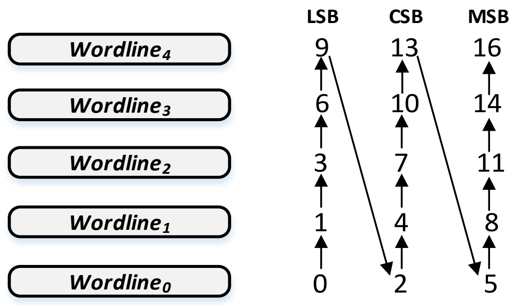

Fig. 8: Relaxed program order in the PAPA scheme.

satisfies the requirement of flexible programming in the PAPA scheme.

- ***Rule 1***: *LSB (i)* must be programmed before *LSB (i+1)*, where i $>$ 0.
- ***Rule 2***: *CSB (i)* must be programmed before *CSB (i+1)*, where i $>$ 0.
- ***Rule 3***: *MSB (i)* must be programmed before *MSB (i+1)*, where i $>$ 0.
- ***Rule 4***: *LSB (i+1)* must be programmed before *CSB (i)*, for all *i*.
- ***Rule 5***: *LSB (i+2)* and *CSB (i+1)* must be programmed before *MSB (i)*, for all *i*.
- ***Rule 6***: *CSB (i)* must be programmed before *LSB (i+2)*, for all *i*.
- ***Rule 7***: *MSB (i)* must be programmed before *LSB (i+3)* and *CSB (i+2)*, for all *i*.

The aforementioned strict rules in the fixed program order are used to minimize the inter-cell interference. ***Rules 1***, ***2***, and ***3*** imply the program order between same page-types, whereas, ***Rules 4*** and ***5*** dictate the program order for different page-types. However, ***Rules 6*** and ***7*** are the over-specifications to mitigate the cell-to-cell interference, and therfore, they can be avoided because programming wordlines *i+2* and *i+3* would not introduce interference in wordline *i*. Park et. al. [8] illustrated a similar scenario for the MLC NAND flash and proved that removing over-specified constraints from the strict program order in the MLC flash, does not increase the program interference in the programmed wordlines. Because both MLC and TLC flash memories share very similar cell-to-cell interference characteristics, our proposed scheme, PAPA, uses a similar relaxed program order by removing aforementioned ***Rules 6*** and ***7***. In addition, the proposed PAPA scheme can be applied to QLC flash, because QLC flash exhibits similar latency asymmetries as that of the TLC flash, among its various types of pages.

C. Relaxed Program Order in PAPA

As discussed in the previous subsection, the strict program order is an over-specification to reduce the program interference. We propose to use a relaxed program order that writes all the LSB, CSB, and MSB pages in a sequence, as depicted in Figure 8. The relaxed program order depicted in Figure 8 obeys all the rules mentioned in the previous subsection except ***Rules 6*** and ***7***, which are over-considered constraints in the conventional TLC program order. Using such relaxed program sequence in the PAPA scheme, we can provide multiple active blocks, with different page-types available in each plane.

In PAPA, two active blocks are simultaneously required in each plane; one block accommodates the partial writes in LSB pages, and the other block stores the full page writes in CSB pages. After the LSB pages are exhausted in the block assigned for partial writes, the next block with free LSB pages is allocated for partial writes. In this manner, the LSB pages of all the blocks are allocated for writing partial pages sequentially. When the LSB pages in all the blocks are consumed, the CSB pages in the partially written blocks,

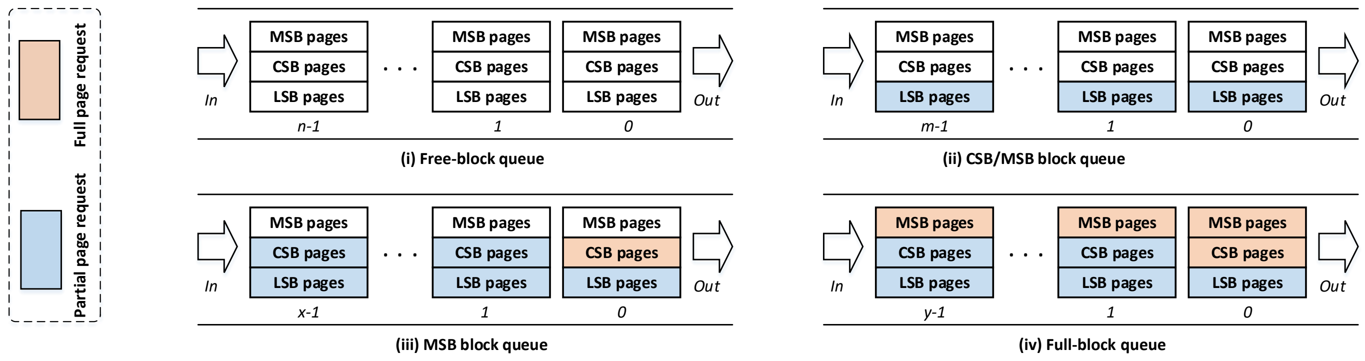

Fig. 9: Block management and life cycle.

whose all LSB pages have been consumed, are used as back up for partial writes. Similarly, we allocate CSB pages of a block whose LSB pages are completely consumed, for full page writes. Once the CSB pages of a block are exhausted, the next block with free CSB pages is allocated for full page writes. However, when there is no block with free CSB pages, the block with free MSB pages is allocated for full page writes. Therefore, the partial pages are always written in LSB/CSB pages and the full pages in CSB/MSB pages. In addition, we maintain the current status of the blocks for achieving efficient block management and page-type assignment, which we will discuss in the following subsection.

D. Block Management and Life Cycle

According to the available free pages, a block can have the following status in our proposed PAPA scheme: free block, CSB/MSB block (block with free CSB and MSB pages), MSB block (block with only free MSB pages), and full block (block with no free pages). By managing the status of the blocks in each plane, we can simplify the allocation of the specific page type in the blocks in each plane. Figure 9 depicts the aforementioned four possible stages of the blocks. Initially, all the blocks are free, i.e., no data written yet, and therefore are kept in the free-block queue (see (i) of Figure 9).

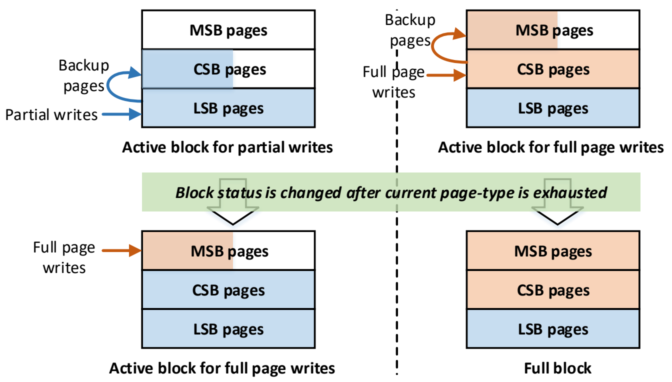
MSB pages
CSB pages
LSB pages
Backup pages
Partial writes
Active block for partial writes
Backup pages
Full page writes
Active block for full page writes
Block status is changed after current page-type is exhausted
Full page writes
Active block for full page writes
Full block

Fig. 10: Writing backup pages and block-status change in the current active block.

Upon receiving a partial page request, the LSB page of *block 0* from the free-block queue is allocated for partial page writing. In addition, all the subsequent partial page writes are written to the LSB pages of *block 0* until the LSB pages of this block are consumed. Please note that we do not maintain a queue for LSB blocks, i.e., blocks with free LSB pages, because a block with free LSB pages suggests that the entire block is free, as LSB pages are written prior to wiritng CSB/MSB pages. Therefore, the block is kept in the free-block queue. Once the LSB pages in the current block are exhausted, the block is enqueued in the CSB/MSB block queue (see (ii) of Figure 9) for servicing full page writes in the CSB pages, and the next block from the free-block queue is allocated for partial writes. The blocks in the CSB/MSB block queue (see (ii) of Figure 9) have free CSB and MSB pages, and therefore, they can be used for full page writes.

Once the CSB pages of a block in the CSB/MSB block queue (see (ii) of Figure 9) are consumed, the block is enqueued in the MSB block queue (see (iii) of Figure 9). When the CSB pages of all the blocks in CSB/MSB block queue (see (ii) of Figure 9) are consumed, the queue becomes empty, and the subsequent full page writes will be written to the MSB pages of the blocks in the MSB block queue (see (iii) of Figure 9). Finally, the blocks with no free pages in the MSB block queue are enqueued in the full-block queue. If a partial page request is received, and the free-block queue and CSB/MSB block queue are empty, an emergency GC is invoked to claim the space for the incoming partial page request. The GC selects a victim block with maximum number of invalid pages from the full-block queue (see (iv) of Fig. 9) and enqueues the block in the free-block queue after cleaning, to accommodate partial writes in the free LSB pages.

Figure 10 depicts a scenario wherein there are no free LSB pages for the incoming partial writes (left side of Figure 10). In this example, we assume that there is no free LSB page in any block. In such case, free CSB pages are considered backup pages for partial writes. As the figure (top left of Figure 10) depicts, partial writes are received and written to the CSB pages of the active block. When the LSB and CSB pages of the active block are exhausted, the status of the block is changed to MSB block, as it has only free MSB pages left for accommodating the full page writes (bottom left of Figure 10).

The right side of Figure 10 depicts another scenario wherein full page requests are received, and the LSB and CSB pages have been exhausted in all the blocks. In such cases, the free MSB pages are written as backup to the CSB pages (top right of Figure 10). After the MSB pages have also been written, the block changes its status to the full block (bottom right of Figure 10). In summary, partial writes can only be written to LSB and CSB pages, and full page writes can only be written to CSB and MSB pages with an exception that full page writes are received before any data are written to the SSD. In this situation, the full pages are written to the LSB pages of the active block, as the CSB pages cannot be written until the LSB pages in the active block are exhausted.

VI. Evaluation

A. Environment Setup

We implemented the proposed PAPA scheme by extending the FlashSim [19] simulator, which is a widely used trace-driven simulation framework for NAND flash storage systems, and its accuracy and behavior have been validated against commercial SSDs. For evaluating the effectiveness of the proposed PAPA scheme, following modifications have been made to the FlashSim simulator:

- LSB, CSB, and MSB pages are assigned different read/write latency values for supporting the diverse read/write latencies in the TLC SSD [2], [3].
- The page-allocation mechanism is modified to support the type-directed page-allocation in the PAPA scheme.
- The traces are run twice, with disabled cache, for observing the long term behavior of the PAPA scheme and for evaluating its impact on garbage collection (GC).

TABLE I: Characteristics of the evaluated I/O traces.

Traces	**Total requests**	**Writes (%)**	**Reads (%)**	**Average size (bytes)**
prxy	1048576	95	5	2421
proj	1048582	76	24	11883
fin	5334987	77	23	4196
mds	1048576	87	13	7523
hm	1048498	73	27	6086
prn	1048579	86	14	12544
tpce	4510214	27	73	17701
tpcc	6286764	67	33	11942
exch	766362	69	31	12768

TABLE II: Configuration of the simulated TLC SSD.

Items	Values	Items	Values
Packages	2	Sector size	4KB
Dies/Packages	2	Page size	16KB
Planes/Die	2	Read latency (μs)	*LSB/CSB/MSB 50/100/150*
Blocks/Plane	2048	Write latency (μs)	*LSB/CSB/MSB 500/2000/5500*
Pages/Block	384	Erase latency (μs)	*3000*

For a comprehensively evaluating the PAPA scheme, 9 real workloads, which were collected from OLTP applications [20] and the Storage Networking Industry Association (SNIA) [21], were used in the experiments. The workload traces are reconstructed with the help of TraceTracker [22] technique that refines the old block I/O traces from OLTP applications [20] and SNIA [21] to make them compatible with modern hardwares. The sector size of these workloads in our experiments is 4KB, which is the standard sector size in modern traces. The key characteristics of these workload traces are listed in Table I. Moreover, the configuration of the simulated TLC SSD is listed in Table II. As our target SSDs are the TLC SSDs with large flash page size, we use 16KB flash page size, as mentioned in Table II. In addition, to compare the effectiveness of PAPA scheme with that of other schemes, we implemented a baseline scheme with the conventional program sequence, an LSB$_{first}$ scheme [3] consuming all the LSB pages prior to writing CSB/MSB pages, and a size-based scheme [3] that assigns LSB pages for the write requests whose size is smaller than the page size and assigns CSB/MSB pages otherwise. Please note that the size-based scheme and the proposed PAPA scheme are distinctively different from each other, as the former only analyzes whether the size of the incoming request from the host is less than the page size, whereas, the latter considers the size of each transaction after the host interface splits the request into page-sized transactions.

B. Experimental Results

*1) **Flash Writes Distribution**:* Figure 11 depicts the ratio of LSB, CSB, and MSB writes in baseline, LSB$_{first}$, size-based, and proposed PAPA schemes. The baseline scheme writes the pages in a fixed program order, and therefore the writes are equally distributed to the three types of pages. The LSB$_{first}$ scheme prioritizes LSB pages for the incoming write requests, and once all the LSB pages are exhausted, it assigns both CSB and MSB pages with equal probability to the future writes. LSB$_{first}$ scheme is effective for workloads with small ratio of write requests, as low-latency LSB pages can accommodate most write requests without consuming a significant number of high-latency CSB/MSB pages. However, if the number of write requests increases, the high-latency CSB/MSB pages are also consumed for accommodating write requests. Therefore, the effectiveness of the LSB$_{first}$ scheme depends considerably on the workload size.

The size-based scheme greedily stores the small writes, i.e., requests with a single transaction, in low-latency LSB pages to decrease the response time, whereas, the high-latency CSB/MSB pages are used to serve large writes, i.e., requests with multiple transactions. Therefore, the utilization of pages by the size-based scheme varies with workload characteristics. If the workloads contain mostly small-writes, the size-based scheme proactively uses LSB pages, thereby increasing the performance of the device. However, GC is triggered frequently to claim LSB pages for accommodating future small-sized writes, thereby reducing both the lifetime and SSD performance.

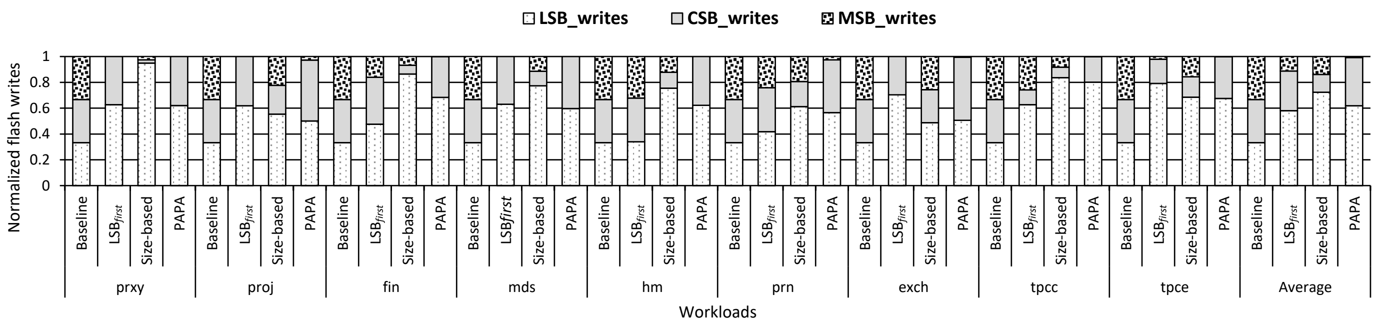

Fig. 11: LSB, CSB, and MSB flash writes in various workloads for different schemes. The flash writes are normalized to the baseline scheme.

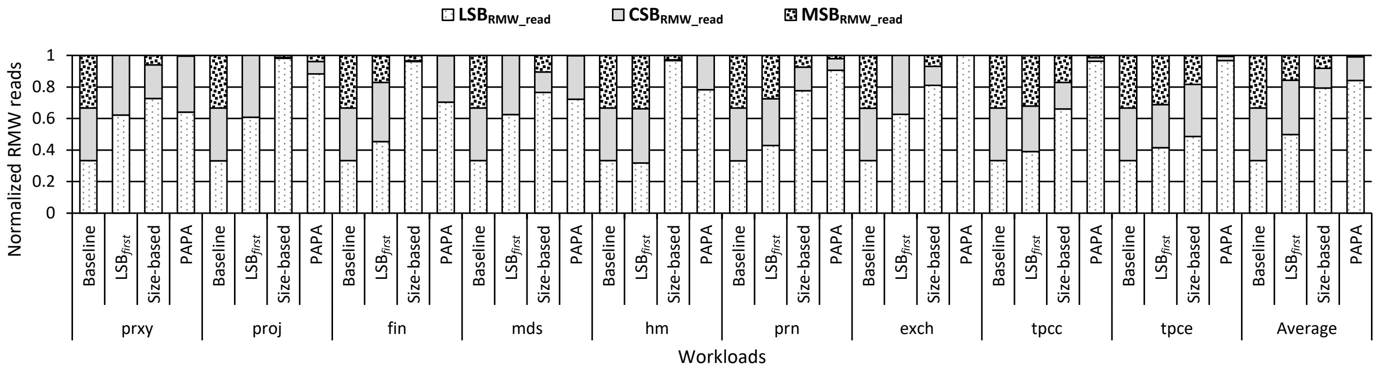

Fig. 12: LSB, CSB, and MSB RMW reads in various workloads for different schemes. The RMW reads are normalized to the baseline scheme.

However, the PAPA scheme preferably allocates LSB pages to partial writes, considering the RMW overhead involved in partial updates. Because modern SSDs have large page sizes, large number of partial pages are created. Therefore, LSB/CSB pages are written more than MSB pages in the PAPA scheme. On average, less than 1% writes are mapped to the MSB pages, thereby significantly reducing the writes and RMW latencies, whereas LSB and CSB pages share 62% and 37.2% writes, respectively.

The impact of reducing the number of writes on high-latency CSB/MSB pages is reflected on the reduced number of RMW reads on these pages. Figure 12 depicts the breakdown of RMW reads on each page type in the evaluated schemes. Similar to flash writes, significant amount of RMW reads are performed on CSB/MSB pages in the LSB$_{first}$ scheme, thereby increasing the RMW latency. On average, 34% and 16% RMW reads are performed on CSB and MSB pages, respectively, in the LSB$_{first}$ scheme. Because the size-based scheme greedily writes small writes to LSB pages, most RMW reads, i.e., 79%, are also performed on these pages. In addition, 13% RMW reads are performed on CSB pages and only 8% on MSB pages.

With the increase in the number of low-latency LSB/CSB page writes, in PAPA, the number of RMW reads on low-latency LSB/CSB page also increases. On average, 84% RMW reads are performed on LSB pages and 15% on CSB pages, whereas MSB pages account for only 1% RMW reads. This type of wise distribution of flash writes and RMW reads to LSB, CSB, and MSB pages, significantly decreases both the write response time and RMW latency of the SSD, which will be discussed in the subsections to follow.

*2) **Impact on lifetime:*** The number of erase operations directly affects the lifetime of the device. Moreover, GC is triggered periodically, whenever the threshold is reached, to clean the invalid space for future writes. The page allocation in the LSB$_{first}$ scheme is similar to that of the baseline, except that the former greedily writes LSB pages prior to writing CSB/MSB pages. Therefore, both the schemes reach the GC threshold at a similar rate, and therefore, both experience nearly equal number of erase operations, as depicted in Figure 13.

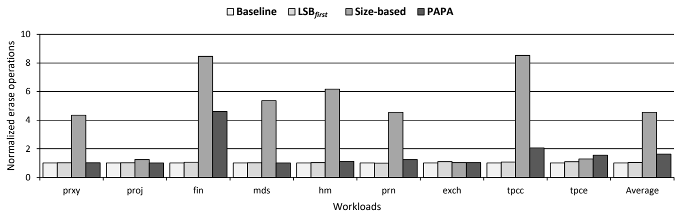

Fig. 13: Normalized erase operations in various workloads for different schemes. The erase operation is normalized to the baseline scheme.

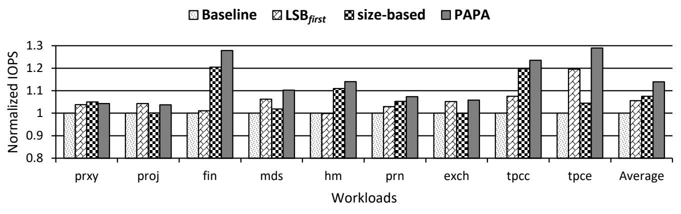

Fig. 14: Normalized I/O throughput (IOPS) in various workloads for different schemes. The I/O throughput (IOPS) is normalized to the baseline scheme.

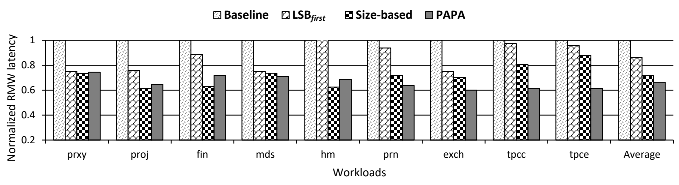

Fig. 16: Normalized RMW latency in various workloads for different schemes. The RMW latency is normalized to the baseline scheme.

The size-based scheme aggressively writes small-writes to LSB pages for performing latency reduction. However, for workloads with mostly small-writes, LSB pages are used up quickly, thereby frequently executing GC to provide more LSB pages for future small-writes. Consequently, an increased number of erase operations significantly degrade the device lifetime. As can be seen in Figure 13, in the worst case, the size-based scheme increases the number of erase operations by more than 8 times in *fin* and *tpcc* traces, owing to the high ratio of small-writes in these workload traces. Whereas, the erase operations are increased by 4.5 times, on average.

However, once LSB pages are consumed, our proposed scheme PAPA uses CSB pages as a backup for partial writes. Consequently, compared with the size-based scheme, PAPA executes GC less frequently, thereby preventing significant lifetime degradation. As depicted in Figure 13, erase operations are performed not more than two times, in most workloads, except that of the *fin* trace. The reason behind the increased number of erase operations in *fin* trace is that it is a write-dominating workload trace with high ratio of partial writes, and therefore, the consumption rate of LSB/CSB pages is considerably higher than that of MSB pages. Hence, more blocks are erased for making LSB/CSB pages available. On average, PAPA increases the number of erase operations by 1.6 times compared to that of the baseline scheme. The increased number of erase operations is the cost of biasing towards LSB pages, as the GC is invoked proactively to ensure the availability of the LSB pages for partial writes.

3) ***Performance improvement****:* Here we report the performance improvements using our proposed scheme, PAPA. We used two metrics, namely, throughput (*IOPS*) and response time, to demonstrate the performance of the evaluated schemes.

Throughout (IOPS). To evaluate and compare the performance improvements obtained using the proposed scheme with those of other evaluated schemes, we measured the SSD throughput *(IOPS)*. The normalized value of the *IOPS* for various schemes is depicted in Figure 14. If the number of LSB/CSB writes is increased and number of MSB writes is reduced, more read/write operations can be performed per unit time. The LSB$_{first}$ scheme greedily assigns LSB pages to the incoming write requests to increase the device throughput. However, after the LSB pages are exhausted, high-latency CSB/MSB pages are used to serve the future writes, thereby degrading the throughput significantly. Therefore, the throughput of the LSB$_{first}$ scheme is different in different phases of the experiment; i.e., the throughput is significantly high in the initial phase when many LSB pages are available, whereas the throughput decreases gradually with decrease in the number of free LSB pages. The *IOPS* in the LSB$_{first}$ scheme is increased by 19% in the best case in the *tpce* trace, owing to the increased number of LSB writes. However, the average increase in *IOPS* in the LSB$_{first}$ scheme is 5.5%.

The size-based scheme aggressively writes small-requests in LSB pages for achieving high throughput, and therefore its performance depends on the number of write requests in the workloads, ratio of small-sized write requests in the workloads, and number of free LSB pages. If the workloads are large-sized with high ratio of small-writes, blocks would be erased frequently for claiming LSB pages to accommodate the future small-writes, thereby negatively affecting the SSD lifetime. Because the evaluated workloads contain mostly small-sized write requests, the size-based scheme could improve *IOPS* significantly, by writing the small-sized write requests to LSB pages. The size-based scheme outperforms other schemes in case of *prxy* trace, owing to the small size of

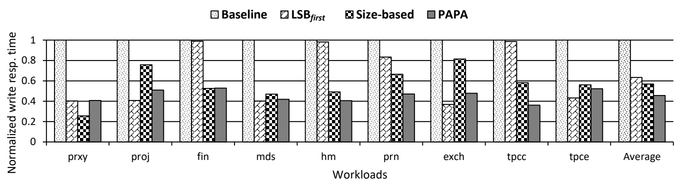

Fig. 15: Normalized write response time in various workloads for different schemes. The write response time is normalized to the baseline scheme.

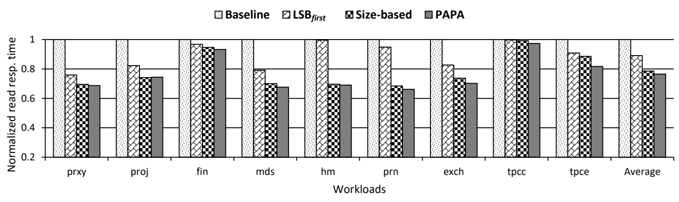

Fig. 17: Normalized read response time in various workloads for different schemes. The read response time is normalized to the baseline scheme.

the workload and high ratio of small-sized write requests. The best improvement in *IOPS* is 20% in *fin* trace, as most the write requests in this trace are small-sized, and therefore, can be written to LSB pages. In addition, the average increase in *IOPS* is 7.5%. Although the size-based scheme effectively improves the SSD throughput, it severely degrades the lifetime because of the frequent execution of GC performed for assuring the availability of free LSB pages (see Section VI-B2).

Our proposed scheme, PAPA, uses low-latency LSB/CSB pages to write partial pages to minimize the writes and RMW latencies. Because the flash page size has been increasing, the ratio of partial pages has also been increasing. Therefore, PAPA leverages low-latency LSB/CSB pages to store partial pages, thereby increasing the SSD throughput. PAPA outperforms the other evaluated schemes in the traces that exhibit high partial page ratio, because of writing the partial writes to LSB pages. The best *IOPS* improvement for PAPA can be seen in the *tpce* trace, which is 29%, because of high partial page ratio in *tpce*. Although *tpce* is not a write-dominating, it is a write-intensive workload with highest number of write requests following *fin* and *tpcc* traces, as shown in Table I. In addition to the large number of write requests, *tpce* also has a significant amount of partial writes and partial updates, as shown in Figure 5. Given these characteristics of *tpce* trace, it eventually performs better than other traces. Moreover, PAPA increases the *IOPS* by 14%, on average, as evident from Figure 14.

Response time. The device response time is another important metric to demonstrate the efficiency of the proposed PAPA scheme. Figure 15 compares the write response times of various schemes. The LSB$_{first}$ scheme greedily assigns LSB pages to the incoming write requests to reduce the write response time. However, after the LSB pages are exhausted, high-latency CSB/MSB pages are used to serve the future writes, thereby increasing the write response time significantly. Therefore, the performance of the LSB$_{first}$ scheme is different in different phases of the experiment; i.e., the performance is significantly high in the initial phase when many LSB pages are available, whereas, it degrades gradually with decrease in the number of free LSB pages. In addition, for large-sized workloads, the performance of the LSB$_{first}$ resembles to that of the baseline scheme, owing to the similar number of LSB/CSB/MSB writes. The write response time of the LSB$_{first}$ scheme is reduced by 63% in the best case in the *exch* trace, owing to the small number of write requests that can be accommodated without writing MSB pages. However, the average reduction in the write response time is 36%.

The size-based scheme aggressively writes small-writes in LSB pages for achieving high throughput, and therefore, its performance considerably depends on the size of the workloads, ratio of small-sized requests in the workloads, and number of free LSB pages. If the workloads are large-sized with high ratio of small-writes, blocks would be erased frequently for claiming LSB pages to accommodate the future small-writes. Because the workloads contain mostly small-writes, the size-based scheme can significantly mitigate the write response time, as apparent from Figure 15. The size-based scheme performs best in case of *prxy* trace, reducing the response time by 75%, owing to the small-sized requests in the *prxy* trace; however, the average reduction in the response time is 43%. In addition, the size-based scheme satisfactorily reduces the RMW latency by 29%, on average, owing to the assignment of small-writes to LSB pages. Although the size-based scheme effectively improves the performance, it considerably degrades the lifetime of the SSD.

Because partial updates incur both additional latency, along with the writes latency, partial pages should be written to low-latency LSB pages. To minimize the aforementioned latencies, the PAPA scheme uses LSB/CSB pages to write partial pages and CSB/MSB pages to write full pages. Because the flash page size has been increasing, the ratio of partial pages has also been increasing. Therefore, PAPA leverages low-latency LSB/CSB pages to store partial pages, thereby reducing both the write response time and RMW latency. The average reduction in the write response time is 55%, and that in the RMW latency is 34%.

To improve the read performance, the read requests are not considered explicitly in the above-mentioned schemes. However, as the above-mentioned schemes attempt to maximize the low-latency LSB writes, the read performance improves implicitly. Figure 17 presents the read response time in various schemes. Both the LSB$_{first}$ and size-based schemes show promising results by reducing the read response time by 11% and 21%, respectively, on average. However, PAPA outperforms the other schemes, as evident from the Figure 17. We attribute this performance improvement of the PAPA scheme to LSB/CSB reads, as most of the data are stored in and read from these pages. The average reduction in the read response time for PAPA is 23%. Therefore, the *IOPS*, write response time, RMW latency, and the read response time improve by 14%, 55%, 34%, and 23%, respectively, as a result of biasing towards low-latency LSB/CSB pages in PAPA.

4) ***Effect of DRAM cache on flash writes:*** The results shown in Section VI-B were obtained by disabling the internal cache to increase the number of partial writes written to the flash memory. However, modern SSDs use a cache for absorbing frequent writes and for reducing the amount of written data to the flash memory. In this subsection, we show the impact of cache on flash writes and argue that the efficiency of our proposed scheme is not affected significantly with the presence of cache. Here we show flash writes distribution among LSB, CSB, and MSB pages, and write response time to demonstrate the impact of cache on the evaluated schemes.

Flash Writes Distribution. Figure 18 depicts the ratio of the LSB, CSB, and MSB writes in the baseline, LSB$_{first}$, size-based, and proposed PAPA schemes. The baseline scheme writes the pages in a fixed program order, and therefore the writes are equally distributed to the three types of pages. The LSB$_{first}$ scheme prioritizes LSB pages for the incoming write requests, and once all the LSB pages are exhausted, it assigns both CSB and MSB pages with equal probability to the future writes. LSB$_{first}$ scheme maximizes the LSB writes in *prxy* and

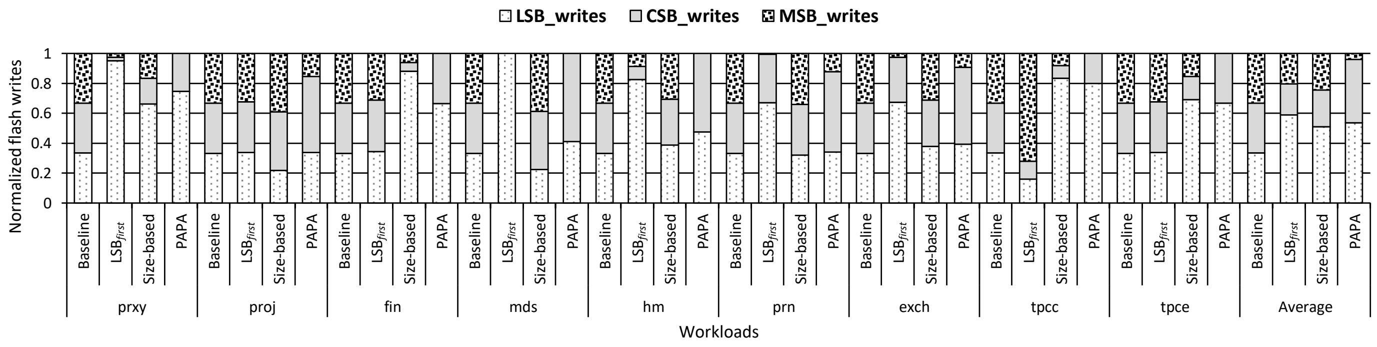

Fig. 18: Flash writes in LSB, CSB, and MSB writes in various workloads for different schemes with cache enabled. The writes are normalized to the baseline scheme.

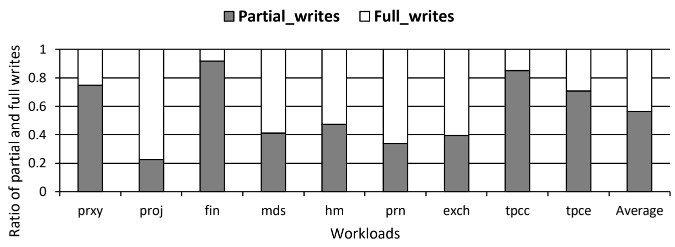

Fig. 19: Ratio of partial and full page writes with cache enabled.

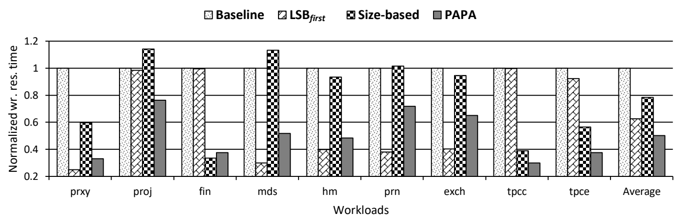

Fig. 20: Normalized write response time in various workloads for different schemes with cache enabled. The write response time is normalized to the baseline scheme.

mds traces, as depicted in Figure 18, owing to small number of writes to flash memory, as many writes are absorbed in the cache in these traces.

The size-based scheme greedily stores the small writes, i.e., requests with a single transaction, in low-latency LSB pages to decrease the response time, whereas, the high-latency CSB/MSB pages are used to serve large writes, i.e., requests with multiple transactions. Therefore, the utilization of pages by the size-based scheme varies with workload characteristics. If the workloads contain mostly small-writes, the size-based scheme proactively uses LSB pages, thereby increasing the performance of the device. However, many single-transaction requests (small requests) are absorbed by the cache, and thus the utilization of LSB pages is reduced in size-based scheme, as depicted in Figure 18.

The PAPA scheme preferably allocates LSB pages to partial writes, considering the RMW overhead involved in partial updates. Although the cache absorbs many partial writes, a significant amount of partial writes are written to the flash memory, as depicted in Figure 19. As shown in the figure, 57% writes are partial writes, whereas, 43% are full page writes. On average, 4% writes are mapped to the MSB pages, thereby significantly reducing the write response time, whereas LSB and CSB pages share 54% and 42% writes, respectively, as depicted in Figure 18.

Write response time. As we discussed in the preceding subsection, the high-latency MSB writes are reduced excessively in PAPA scheme. Therefore, the write response time is also improved significantly, as depicted in Figure 20. The best improvement in write response time can be seen in *tpcc* trace for PAPA, owing the large number of partial writes which are written to low-latency LSB/CSB pages. However, the average reduction in write response time in PAPA is 50%, as depicted in Figure 20. To conclude, our proposed scheme, PAPA, maximizes the low-latency LSB writes and minimizes the high-latency MSB writes and improves the response time effectively both in presence and absence of the internal cache.

VII. Conclusion

Different types of pages in the TLC flash memory exhibit variable read/write latencies. In addition, partial updates perform worst when updating an MSB page, as partial updates perform an additional read operation for ensuring data integrity. Conventional TLC programming designs follow a type-blind page allocation, thereby providing an opportunity to explore and utilize a new type-directed page allocation scheme for performance enhancement. In this study, we proposed the PAPA scheme for the TLC flash memory. The proposed scheme simultaneously considers the size of each transaction and flash page types for performing page allocation. We employed a relaxed program order to meet the requirements of our type-directed page allocation. The PAPA scheme prioritizes low-latency LSB pages for partial page writes and high-latency CSB/MSB pages for full page writes. We evaluated the performance of the PAPA scheme using a rich set of workload traces. We observed that the SSD throughput, write response time, and RMW latency improved by 14%, 55%, and 34%, on average, respectively.

References

[1] D. Sharma, "System design for mainstream tlc ssd meeting the performance challenge," in *Proc. Flash Memory SUMMIT*, 2014, pp. 1–20.

[2] W. Choi, M. Jung, and M. Kandemir, "Invalid data-aware coding to enhance the read performance of high-density flash memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 482–493.

[3] W. Zhang, Q. Cao, H. Jiang, and J. Yao, "Improving overall performance of tlc ssd by exploiting dissimilarity of flash pages," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[4] L. M. Grupp, J. D. Davis, and S. Swanson, "The harey tortoise: managing heterogeneous write performance in ssds," in *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 79–90.

[5] S. Jin, J. Kim, J. Kim, J. Huh, and S. Maeng, "Sector log: fine-grained storage management for solid state drives," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 360–367.

[6] M. Kang, W. Lee, and S. Kim, "Subpage-aware solid state drive for improving lifetime and performance," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1492–1505, 2018.

[7] Y. Feng, D. Feng, C. Yu, W. Tong, and J. Liu, "Mapping granularity adaptive ftl based on flash page re-programming," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 374–379.

[8] J. Park, J. Jeong, S. Lee, Y. Song, and J. Kim, "Improving performance and lifetime of nand storage systems using relaxed program sequence," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[9] A. Tavakkol, P. Mehrvarzy, M. Arjomand, and H. Sarbazi-Azad, "Performance evaluation of dynamic page allocation strategies in ssds," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 2, p. 7, 2016.

[10] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[11] C.-W. Chang, G.-Y. Chen, Y.-J. Chen, C.-W. Yeh, P. Y. Eng, A. Cheung, and C.-L. Yang, "Exploiting write heterogeneity of morphable mlc/slc ssds in datacenters with service-level objectives," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1457–1463, 2017.

[12] Y. Li, C. Hsu, and K. Oowada, "Non-volatile memory and method with improved first pass programming," Aug. 19 2014, uS Patent 8,811,091.

[13] M. Murugan and D. H. Du, "Hybrot: Towards improved performance in hybrid slc-mlc devices," in *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2012, pp. 481–484.

[14] L.-P. Chang, "A hybrid approach to nand-flash-based solid-state disks," *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1337–1349, 2010.

[15] S. Im and D. Shin, "Comboftl: Improving performance and lifespan of mlc flash memory using slc flash buffer," *Journal of Systems Architecture*, vol. 56, no. 12, pp. 641–653, 2010.

[16] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim, "Flexfs: A flexible flash file system for mlc nand flash memory." in *USENIX annual technical conference*, 2009, pp. 1–14.

[17] W. Wang, W. Pan, T. Xie, and D. Zhou, "How many mlcs should impersonate slcs to optimize ssd performance?" in *Proceedings of the Second International Symposium on Memory Systems*. ACM, 2016, pp. 238–247.

[18] 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*. ACM, 2011, pp. 96–107.

[19] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *2009 First International Conference on Advances in System Simulation*. IEEE, 2009, pp. 125–131.

[20] U. trace repository, "Umass trace repository." [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[21] S. I. Repository, "Snia repository." [Online]. Available: http://iotta.snia.org/traces/130

[22] M. Kwon, J. Zhang, G. Park, W. Choi, D. Donofrio, J. Shalf, M. Kandemir, and M. Jung, "Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2017, pp. 87–96.