SUPA: A Single Unified Read-Write Buffer and Pattern-Change-Aware FTL for the High Performance of Multi-Channel SSD

DONGJIN KIM, KYU HO PARK, and CHAN-HYUN YOUN, KAIST

To design the write buffer and flash translation layer (FTL) for a solid-state drive (SSD), previous studies have tried to increase overall SSD performance by parallel I/O and garbage collection overhead reduction. Recent works have proposed pattern-based managements, which uses the request size and read- or write-intensiveness to apply different policies to each type of data. In our observation, the locations of read and write requests are closely related, and the pattern of each type of data can be changed. In this work, we propose SUPA, a single unified read-write buffer and pattern-change-aware FTL on multi-channel SSD architecture. To increase both read and write hit ratios on the buffer based on locality, we use a single unified read-write buffer for both clean and dirty blocks. With proposed buffer, we can increase buffer hit ratio up to 8.0% and reduce 33.6% and 7.5% of read and write latencies, respectively. To handle pattern-changed blocks, we add a pattern handler between the buffer and the FTL, which monitors channel status and handles data by applying one of the two different policies according to the pattern changes. With pattern change handling process, we can reduce 1.0% and 15.4% of read and write latencies, respectively. In total, our evaluations show that SUPA can get up to 2.0 and 3.9 times less read and write latency, respectively, without loss of lifetime in comparison to previous works.

CCS Concepts: • **Information systems** → *Storage management*; • **Software and its engineering** → *Secondary storage*;

Additional Key Words and Phrases: Multi-channel SSD, NAND flash memory, Read-write unified buffer, Pattern-based management, Pattern handler, Flash Translation Layer

ACM Reference format:
Dongjin Kim, Kyu Ho Park, and Chan-Hyun Youn. 2017. SUPA: A Single Unified Read-Write Buffer and Pattern-Change-Aware FTL for the High Performance of Multi-Channel SSD. *ACM Trans. Storage* 13, 4, Article 32 (November 2017), 30 pages.
https://doi.org/10.1145/3129901

1 INTRODUCTION

There have been many studies to increase SSD performance through enhancement of the write buffer (e.g., BPLRU [15] and CAVE [31]) and flash translation layer (FTL) (e.g., DFTL [9] and MNK [38]). They have focused on increasing parallel I/O and reducing garbage collection (GC)

This work was supported by 'The Cross-Ministry Giga KOREA Project' grant funded by the Korea government(MSIT) (No.GK17P0100, Development of Tele-Experience Service SW Platform based on Giga Media) and the National Research Council of Science & Technology(NST) grant by the Korea government (MSIT) (No. CAP-17-03-KISTI).
Authors' addresses: D. Kim, K. H. Park, and C.-H. Youn, School of Electrical Engineering, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea; emails: {dongjinkim, kpark, chyoun}@kaist.ac.kr.

https://doi.org/10.1145/3129901

overhead. By choosing appropriate locations among NAND flash memory banks, I/O performance can be maximized due to parallel access. There have also been efforts to reduce GC overhead, which is unavoidable to reuse some parts of an SSD.

In particular, recent works have classified patterns of blocks. Blocks requested from the host are divided into sequential and random patterns, or read-intensive and write-intensive patterns. Sequential and random classification is done based on the length of the write request. If the length is long enough, then corresponding blocks are classified as sequential. If it is short, then corresponding blocks are classified as random [30]. Read- or write-intensiveness is determined by the number of continuous read requests between two write requests for a block. A block is classified as a write-intensive block when a write request comes, and it becomes read-intensive if the enough number of continuous read requests come before a new write request [4]. For blocks with different patterns, different mapping policies are required. For sequential or read-intensive blocks, a mapping policy tries to increase parallel access by scattering pages or blocks into several channels. A random or write-intensive block tries to reduce the GC overhead through allocation into a single channel.

Beyond previous works, there are still issues regarding performance. First, read and write requests have a locality that is closely related to the SSD performance. In a typical write buffer, read requests for pages existing in the buffer can be directly returned to the host. For non-existing pages, unlike write requests, pages read from NAND flash memory banks are not inserted into the buffer. In real workloads, however, read and write requests can occur in the same location. Many pages are accessed by both read and write requests while they stay in the buffer. Thus, handling read and write requests in the same buffer can enhance not only read latency but also write latency.

Additionally, the pattern of a block can be changed between sequential and random or read-intensive and write-intensive, and this can increase the I/O latency due to an old policy application or a policy switching. When a block shows a pattern change, if an appropriate policy for an incoming pattern is not applied, it will be handled by an old policy that can increase latencies; handling sequential blocks with a single channel will suffer lack of parallel access, and handling random blocks with multiple channels will cause more GC overhead. To avoid these overheads, handling policy will switch to the policy based on an incoming pattern, and this switching can incur extra read and write operations, since physical locations of requested pages will be changed. Therefore, additional handling of blocks that show pattern change is needed.

Through these motivations, we propose a single unified read-write buffer and pattern-change-aware FTL (SUPA) for high performance of multi-channel SSD architecture. The contributions of this study are summarized in the following text.

- **A single unified read-write buffer**: Since read and write requests can occur in the same location, SUPA uses a single unified read-write buffer. It manages both clean and dirty blocks with a same priority in a single list. The buffer classifies patterns of blocks at just before a victim selection process. Based on the length of contiguous read-accessed or write-accessed pages, each block will be classified sequential or random for both read and write patterns. Victims for read requests and write requests are selected in different ways based on the properties of NAND flashes. When an eviction is occurred by a read miss, clean blocks are selected first as victims not to increase the read latency by write operations. In case of an eviction by a write miss, multiple clean and dirty victims will be selected to be written in parallel. Compared to a typical write buffer, our buffer can increase the buffer hit ratios up to 8.0%, and it can reduce 33.6% and 7.5% of read and write latencies, respectively.
- **Pattern-change-aware FTL**: To reduce the pattern change overheads, SUPA uses a layer called a pattern handler. It receives access requests from the buffer, and sends actual read

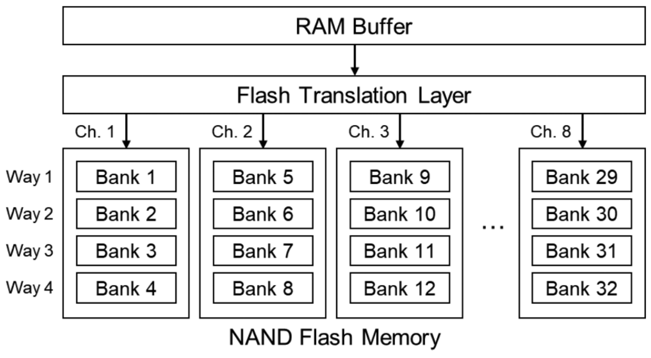


Fig. 1. Typical internal SSD architecture.

and write requests to FTLs. It also detects pattern changes and handle them. When a pattern change is detected, the pattern handler induces pattern-changed data to be stored in a hybrid of multiple FTL policies for different patterns, until there is no more pattern changes for the corresponding blocks. By applying this scheme, we can get 1.0% and 15.4% read and write latency reductions, respectively.

The remainder of this article is organized as follows. We give some background of typical SSDs in Section 2. In Section 3, previous works related to the write buffer, FTL, and pattern-based management are introduced. Section 4 presents our motivations. The architecture and operation of SUPA are described in detail in Section 5. We evaluate our work in Section 6, and we conclude in Section 7.

2 BACKGROUND

Since SUPA enhances performance from the typical SSD, we briefly summarize basic structure and operations of the typical SSD. It comprises three main parts as shown in Figure 1: a RAM write buffer, an FTL, and multiple NAND flash memory banks [31, 38]. An SSD has an amount of DRAM or SDRAM inside, and the write buffer and the FTL are located there.

The write buffer reduces small frequent write requests that can lead to increased high GC overhead. Based on spatial and temporal localities, blocks for write requests are written and reordered in the write buffer until they are evicted during the victim selection process, so the write buffer can reduce latencies and GC overhead.

The FTL translates a logical address of a file system to a physical address of NAND flash memory. It also re-allocates the new physical address for rewritten data to avoid in-place update that leads to erase operation, and collects and erases invalid pages as a unit of block. This is called a garbage collection (GC) operation. Since the time taken for the erase operation is strongly related to the overall performance of the SSD, the FTL tries to reduce GC overhead. Additionally, because of the limitation of NAND flash memory erase counts [6], the FTL makes a balance of the erase counts of every NAND flash memory banks, which is called a wear-leveling.

NAND flash memory has non-volatility, reliability, low-power consumption, and shock resistance characteristics [31]. It has multiple blocks, and each block contains multiple pages. Read and write operations are done in page units, while erase operations are carried out in block units. For the storage capacity increment, an SSD has multiple NAND flash memory banks categorized by channel and way, and it is possible to access several banks in parallel.

3 RELATED WORKS

SSDs have been widely researched in several areas. We briefly summarize previous works about write buffer management, FTL algorithm, and pattern-based management of SSDs.

3.1 Write Buffer Management

Write buffer management schemes tried to reduce the actual write count on NAND flash memory. CFLRU [32] evicts clean pages first. BPLRU [15] manages an LRU list as a unit of block. FAB [12], REF [35], CLC [14], and BPAC [47] has been employed to take advantage of temporal and spatial localities and consider NAND flash characteristics such as erase-before-write and block-level erase to improve overall performance by reducing GC overhead.

For write buffers in a multi-channel architecture, MCA [36], GB [7], and FAP-LRU [43] try to take advantages of multiple access. In CAVE [31], multiple victim blocks are selected, as many as the number of channels, and each victim is allocated to a single channel.

Some works tried to use non-volatile memory in the write buffer area. In CBM's buffer [44], dirty data are stored in non-volatile memory area, while clean data are stored in DRAM area. Han et al. [10] use hierarchical DRAM and non-volatile memory buffer to handle clean and dirty data. From DRAM buffer, dirty data are evicted first, and they are sent to non-volatile memory area. Evicted data from non-volatile memory are written in the flash area.

These buffering schemes can increase SSD performance. In SUPA, we try to get the better performance with the single unified buffer for both read and write requests, based on a locality of read and write requests.

3.2 FTL Algorithm

FTL algorithms are employed to overcome erase-before-write characteristics. LazyFTL [24] and DFTL [9] use page-level mapping, which translates logical addresses to physical addresses as a unit of page, and it requires a large mapping table. BAST [16], superblock [13], AFTL [46], SAST [27], FAST [21], KAST [8], FASTer [23], and ADAPT [42] use combinations of page-level and block-level mapping. They manage data blocks with block-level mapping and log blocks with page-level mapping. These mapping policies can reduce GC overhead by storing updated data in log blocks.

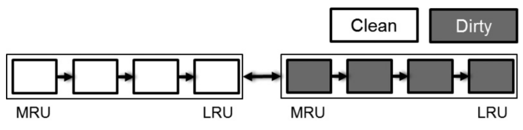
For an FTL in a multi-channel architecture, MCSplit [17], SubGroup [29], and MAST [39] apply hybrid mapping with striping to achieve I/O parallelism. MNK [38] can configure M, N, and K for striping level, the number of data blocks in a group, and the number of log blocks in a group, respectively. VBP-FAST [11] expands parallel I/O to all NAND flash level. DPA-FTL [19] stores hot data in SLC and cold data in MLC. ASA-FTL [48] uses selective caching and sampling for hot and cold separation. Parallel-DFTL [49] schedules address translation and data access operations separately.

These FTLs can achieve GC overhead reduction and better wear-leveling through NAND flash memory banks or channels. Additionally, pattern classification of blocks and the use of different FTL configurations for each pattern will increase the efficiency of the FTL, which are described in Section 3.3.

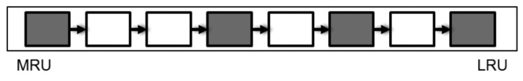
3.3 Pattern-Based Management of SSD

HSSD [5], LAST [20], CFTL [28], and WAFTL [45] separate hot and cold data, or random and sequential data, and manage each pattern in different ways.

For multi-channel architecture, S-FTL [30] divides sequential and random blocks based on the lengths of the write requests at the entrance of the write buffer, and it uses a different write buffer and a different FTL for each pattern. Each block is stored in a single channel without page-level



(a) Separated buffer for clean and dirty blocks



(b) Unified buffer for clean and dirty blocks

Fig. 2. Buffer types for both clean and dirty blocks.

striping, and multiple sequential blocks are grouped into a changed block to be written in parallel channel.

BLAS [4] stores a block in a single channel first, and scatters the pages of that block into several channels if that block is read-intensive or sequential. A block is classified as read-intensive when continuous read requests come into that block, and a block is classified as write-intensive when that block is evicted from the write buffer.

Pattern-based CAVE, a combination of the CAVE write buffer [31] and MNK FTL [38], can also be used for pattern-based management. Based on the numbers of dirty pages, it classifies patterns of blocks. A pattern of a block is classified as sequential if the block has a larger number of pages than the threshold value, or it may be classified as random. Here, multiple victim blocks with random patterns can be selected and a single channel is allocated to each block as CAVE. On the other hand, a victim block with a sequential pattern is scattered over all channels with page-level striping of MNK configuration.

These pattern-based managements can improve the performance of the conventional FTLs described in Section 3.2. In SUPA, we additionally consider pattern changes between sequential and random, or read-intensive and write-intensive.

4 MOTIVATION

Although previous works have improved SSD performance, there are issues for increasing the performance. First, read requests can affect buffer management for increasing the buffer hit ratio and the performance, based on a locality of read and write requests. Additionally, overhead can be caused by blocks' changing pattern.

4.1 Unified Read-Write Buffering

To manage the buffer for read and write requests, there are several types of buffer: a separated buffer with two linked lists for clean and dirty blocks [44] (Figure 2(a)), a unified buffer with a single linked list [32] (Figure 2(b)), or a combination of them. The dirty-only buffer [4, 30, 31], which is a typical write buffer, can be treated as a special case of separated buffer in which buffer size of clean blocks is zero.

Typical write buffers contain only dirty blocks for write requests, and clean blocks for read requests are not inserted into the buffer. Clean blocks, however, are also closely related to the buffer hit ratio and SSD performance. For read requests, since they should be returned directly

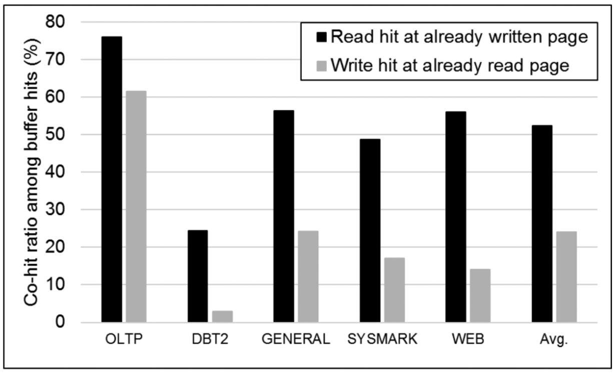


Fig. 3. Co-hit ratios in the unified read-write buffer.

to the host while write requests can be handled in the background, clean block management can increase the read performance.

For write requests, if clean and dirty blocks are independent, write hit ratio will be decreased, since clean blocks will take buffer space of dirty blocks. In read SSD usage, however, read and write requests can occur at the same pages, so read and write hit locations are dependent. Let us define a co-hit as a read hit at an already written page in the buffer, or vice versa. Figure 3 shows how many buffer hits occurred as co-hit for several workloads on the unified buffer. A trace-driven simulator was used [31, 38, 39] with the 64MB unified read-write buffer and the workloads shown in Table 4, which are described in Section 6.1. The sizes of a page and a block were 4 and 256KB, respectively. Each workload had significant amounts of co-hits. On average, 52.3% of read hits occurred at already written pages, and 24.0% of write hits occurred at already read pages. Due to co-hits, if the buffer manages clean and dirty blocks together, it can find more accurate list of recently or frequently accessed pages or blocks, and write hit ratios can be increased, since the corresponding pages more likely exist in the buffer. Thus, we use a single unified read-write buffer for both clean and dirty blocks. Additionally, we adopt different eviction policies for read and write requests based on read and write properties of the NAND flash memory and multi-channel parallelism.

4.2 Pattern Change Handling Overhead

The pattern-based management schemes shown in Section 3.3 handle different patterns by different policies. The pattern of each block, however, can be changed depending on the workload. In the user-level usage, when a file is deleted and a new file is written at the same location, or only a part of a file is modified, corresponding blocks can have a new pattern. Some applications make blocks suffer from frequent pattern changes. For a database application, for example, depending on the query type, a pattern can be changed between sequential and random, or read-intensive and write-intensive.

When the pattern of a block changes, additional overhead can be incurred by an old policy application or a policy switching. When a pattern change occurs for a block, if that block remains in the previous policy, it will be handled by an old policy. Handling sequential blocks with a single channel will suffer lack of parallel access, and handling random blocks with multiple channels will cause more GC overhead. To use an appropriate policy for incoming pattern, switching to a corresponding policy can incur extra read and write operations, since physical locations of

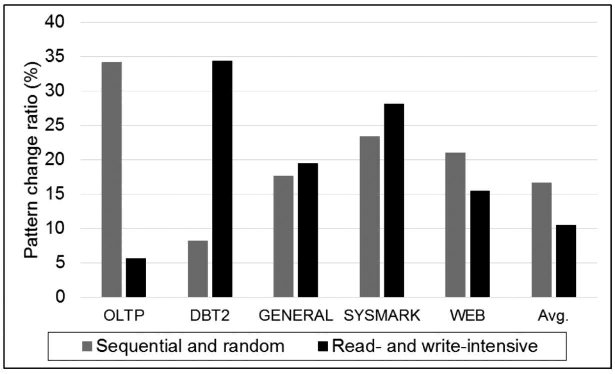


Fig. 4. Pattern change ratios between sequential and random (pattern-based CAVE) and read- and write-intensive (BLAS).

requested pages will be changed. Previous pattern-based managements, however, did not handle this overhead.

Figure 4 shows how many pattern changes occurred in workloads, in the same environment as Section 4.1. A pattern was classified when a block needed to be accessed in the FTL, which were evicted blocks from the write buffer or blocks with buffer miss. Pattern classification thresholds were 4 and 2 for sequential and random division, and read- and write-intensive division, respectively. Here, pattern change includes both an old policy application and a policy switching. On average, 16.6% and 10.4% of block accesses occur pattern change between sequential and random, and read- and write-intensive, respectively. Based on these observations, therefore, we classified both sequential and random patterns and read- and write-intensive patterns, and used different handling policies for blocks that showed pattern change to reduce the overhead.

4.3 Summary

Through these motivations, read and write requests are handled in a unified buffer. We also handle pattern changes both between sequential and random and between read-intensive and write-intensive to reduce the policy switching overhead. Blocks showing pattern change are handled by a hybrid allocation policy.

5 DESIGN OF SUPA

5.1 Overview

In this article, we propose SUPA, a single unified read-write buffer and pattern-change-aware FTL, to enhance the performance of multi-channel SSD. Figure 5 shows the architecture of SUPA, which includes a unified read-write buffer, a pattern handler, and FTL policies for page-level striping on multiple channels and just single channel allocation. The buffer will classify patterns of blocks and evict victims. The pattern handler will monitor pattern-changed blocks and handle them. Their operations are summarized in the following text.

- **Unified read-write buffer**: Each I/O request from the host comes into the buffer. Since locations of read and write requests are closely related, we manage both clean blocks and dirty blocks within the single unified buffer. For both read and write requests, corresponding blocks will be moved to MRU position, while read requests do not move corresponding blocks in typical write buffers.

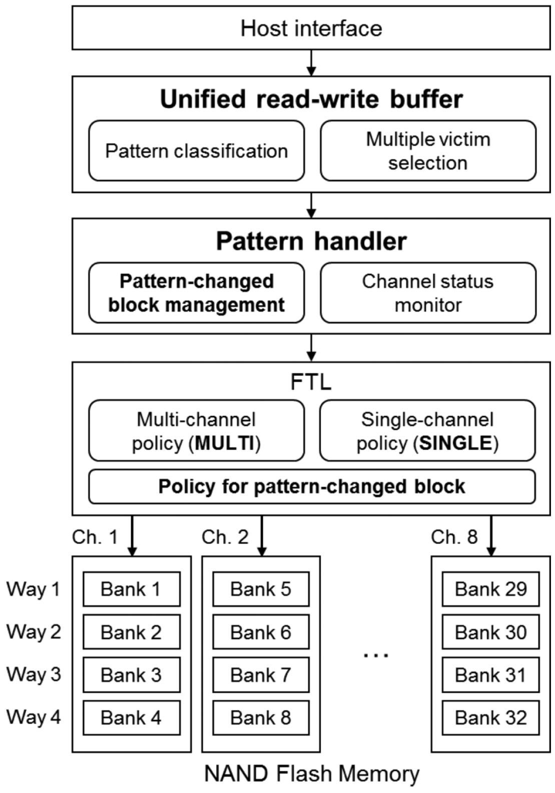


Fig. 5. SUPA overview.

- **Pattern classification**: The buffer classifies patterns of blocks just before a victim selection process. Each block will be classified between sequential and random for both read and write patterns. For read pattern, if the length of contiguous read-accessed pages of a block is long enough to take benefit from parallel access, that block will be classified as sequential, or random if the length is short. Write pattern will be determined by the same manner with write-accessed pages.
- **Victim selection**: Victims are selected in different ways for read and write requests, since properties of read and write operation in the NAND flash memory are different. On a victim selection for a read request, clean blocks will be selected first, since writing dirty victims for read requests can increase the read latency. On a victim selection for a write request, it first selects victim blocks as an amount of requested data and then selects additional victim blocks if they can be written in parallel with other victims. Here, actual writing operations in the NAND flash area will be done in background if flashes are not busy, or waiting time can be occurred if they are busy.
- **Pattern handler**: Before victim blocks go into the FTL, the pattern handler, which is added between the buffer and FTLs, checks whether there has been a pattern change or not. If the current policy for the previous pattern and the new policy for the incoming pattern are different, then a corresponding block is handled as a pattern-changed block. Blocks without pattern change are sent to the FTL directly. For pattern-changed blocks, however, the pattern handler applies a hybrid policy of previous and incoming patterns to them.

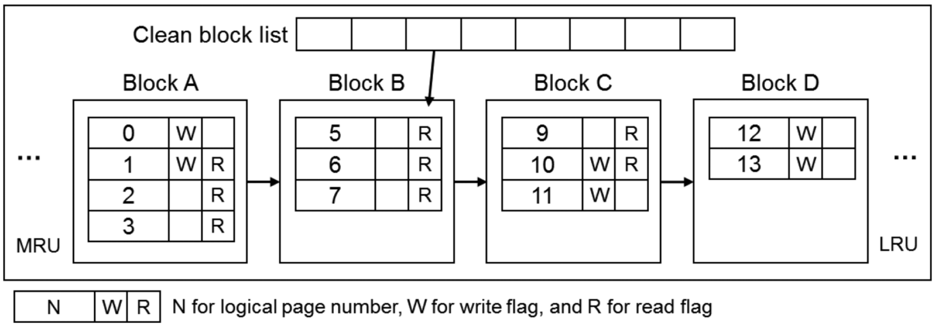


Fig. 6. Buffer structure of SUPA with write and read flags.

- **FTL**: Two policies in the FTL are configured for patterns with different striping levels. A multi-channel policy, MULTI, stores sequential or read-intensive blocks with page-level striping for parallel access, while a single-channel policy, SINGLE, stores a single random or write-intensive block in a single channel to decrease GC overhead. Pattern-changed blocks are stored in a hybrid policy of MULTI and SINGLE. In the hybrid policy, incoming dirty pages are stored in a new policy while others remain in a previous policy. If there are no more pattern changes for those blocks, then all pages will be stored in a recent policy at next GC process.

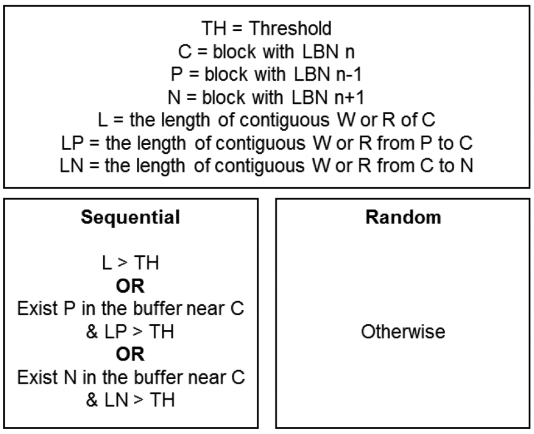
In the remaining parts of this section, components of SUPA are described in detail: the unified read-write buffer in Section 5.2, pattern classification in Section 5.3.1, victim selection in Section 5.3.2 and 5.3.3, and the pattern handler and the FTL in Section 5.4.

5.2 Unified Read-Write Buffer

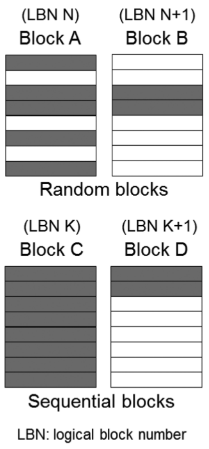
Since read and write hit locations are dependent on each other as described in Section 4.1, SUPA uses a unified read-write buffer. Basically, blocks are managed by a block-level LRU policy [15], because patterns are classified as a unit of block. Additionally, while blocks in the S-FTL [30] are divided into sequential and random buffers, we do not divide them, since a pattern can also be changed in the buffer. Here, pattern classification is not done at the buffer entrance but at the eviction process.

Figure 6 shows the buffer structure. Like typical write buffers, all blocks in the buffer are managed with single linked list. Each block manages its page slots with an internal linked list, and only accessed pages occupy the buffer area. As shown in that figure, blocks A, B, C, and D occupy 4, 3, 3, and 2 pages of the buffer, respectively, which are the same numbers as their accessed pages. For SUPA, it additionally manages a clean block list, which contains pointers of clean blocks for fast clean block searching in the eviction process. For each page, two flags are added: a write flag and a read flag. The write and read flags are set when a write and read request to the page has arrived, respectively. Since only accessed pages can exist in the buffer, at least one of the flags of each page should be set. In Figure 6, only read requests occur for pages in block B while only write requests occur for pages in block D. In blocks A and C, both read and write requests occur for pages 1 and 10. These flags are cleared only when the block is evicted from the buffer. The flags will be used in pattern classification process described in Section 5.3.1.

A typical write buffer and a unified read-write buffer handle write requests in the same manner. For a write request, a corresponding block is moved or inserted to the most-recently-used (MRU) position. On the other hand, two buffers handle read requests differently. For a read request,


TH = Threshold
C = block with LBN n
P = block with LBN n-1
N = block with LBN n+1
L = the length of contiguous W or R of C
LP = the length of contiguous W or R from P to C
LN = the length of contiguous W or R from C to N
Sequential
L > TH
OR
Exist P in the buffer near C & LP > TH
OR
Exist N in the buffer near C & LN > TH
Random
Otherwise

(a) Conditions of sequential and random patterns



(b) Example for contiguous blocks

Fig. 7. Sequential and random pattern classification.

corresponding pages are returned first in both buffers, regardless of buffer hit or miss. In the unified buffer, a corresponding block is moved or inserted to the MRU position as write request case, while the typical write buffer does not change locations of blocks in the buffer. During these processes, if there are not enough page slots for buffer miss cases, the buffer launches the victim selection process with the eviction policy, and incoming pages will take possessions of the secured slots.

5.3 Victim Selection and Eviction of the Buffer

In the victim selection process, the patterns of blocks are classified. After this classification, one or several blocks are evicted, and their pattern information and dirty pages are sent to the pattern handler. Here, SUPA uses different victim selection policies for read and write requests, according to the following reasons. At first, since write operations are slower than read operations in the NAND flash memory [37], dirty victims for read requests can increase the read latency. Additionally, if there are only a few read hit counts or co-hit counts, write hit counts can be reduced. This can happen depending on the workload or host-side buffer. Finally, parallel access should be considered when dirty blocks are evicted.

5.3.1 Pattern Classification. Patterns are classified as sequential or random according to the length of contiguous accessed pages. Figure 7(a) shows how sequential and random blocks are classified. The write pattern of a block is determined based on the length of contiguous write flags. If a block has more contiguous write flags than the threshold value (TH), then the write pattern of that block is classified as sequential, or it is classified as random if the length is same or less. The length of contiguous flags can be measured within the block (L), from previous block (LP), or to next block (LN). Here, since accessing previous or next block in the NAND flash area takes a lot of time, LP and LN are only measured if the corresponding blocks exist in the buffer. The same policy will be used for read pattern with read flags. Here, we experimentally selected the threshold value to minimize the read and write latencies, as it will be mentioned in Section 6.3.1.

Figure 7(b) shows pattern classification examples, where each block contains 8 pages and the threshold value is 4. Block A has more flags than the threshold value, but it is classified as random block. Since accessed pages are scattered within that block, its pattern is more close to random.

Table 1. Pattern Classification

		Write pattern		
		No write	Ran. write	Seq. write
Read pattern	No read (WI)	X	RW	SW
	Ran. read (WI)	RR	RW-RR	SW-RR
	Seq. read (RI)	SR	RW-SR	SW-SR

X, Cannot exist; WI, Write-Intensive; RI, Read-Intensive;
RW, Random Write; SW, Sequential Write;
RR, Random Read; SR, Sequential Read.

Block B and C are obviously random and sequential block, respectively. Block D has only two flags, but it can be contiguous data from previous block (block C), so it is classified as sequential block.

To classify read-intensive and write-intensive blocks, we simply select blocks that have enough read accessed pages regardless of write accessed pages, since they can take benefit from parallel access from page-level striping. Finally, each block can have a pattern among 8 in Table 1, where RW, SW, RR, and SR stand for random write, sequential write, random read, and sequential read, respectively. Since not-accessed block cannot exist in the buffer, a pattern for blocks with no read and no write access cannot occur, which is marked as X in the table. Here, SR, RW-SR, and SW-SR will be treated as read-intensive blocks; otherwise, blocks are write-intensive except RR. Because RR blocks have only a few accessed pages, they are not considered to read-intensive or write-intensive. Based on this pattern classification, pattern changes are defined by the pattern handler described in Section 5.4.

5.3.2 Victim Selection for Read Request. When an eviction from the buffer is needed to handle an incoming read request, clean blocks are mainly selected. This prioritization can avoid writing dirty victim blocks on the FTL for a read request. Additionally, since dirty blocks can stay in the buffer, it can also prevent reduction of the write hit ratio caused by low read hit ratio or low co-hit ratio. During the eviction, clean blocks are selected first until enough space is secured or there are no more clean blocks. After the eviction of clean blocks, if there is enough space for incoming pages, they occupy that space. If not, then one or some dirty blocks are selected near the LRU position, whose the entire number of pages is just fit to the number of incoming pages. After an eviction, dirty pages and the pattern information of victim blocks are sent to the pattern handler so the data can be stored and the pattern change can be checked.

An example of victim selection for a read request is shown in Figure 8. If a read request for two pages in block I comes, then block E will be selected, since it is a clean block and blocks H, G, and F are dirty blocks. Here, because eviction of block E can provide enough space in the buffer, block I can be inserted into the buffer. If block E is a dirty block, then block B will be evicted, which is the next clean block.

5.3.3 Victim Selection for Write Request. On the victim selection for write requests, we select both clean and dirty victim blocks. If clean blocks are mainly selected even for the write request, then a probability to evict dirty blocks for read request can be increased, since only a few clean blocks will remain. Additionally, for parallelized write, several dirty victim blocks can be selected to allocate write operation to multiple channels. Here, we do not allocate blocks for a channel without enough free blocks, since the garbage collection can delay entire write request handling. It can also guarantee the write performance when single-channel writes delay multi-channel writes, since multiple single-channel writes can also be done in parallel. The victim selection for the write

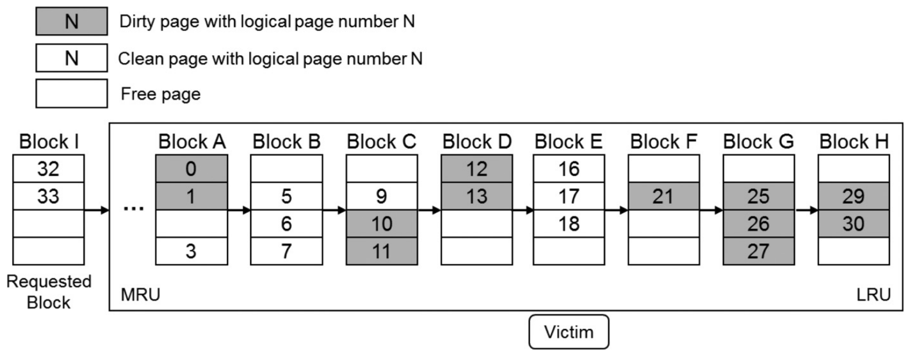


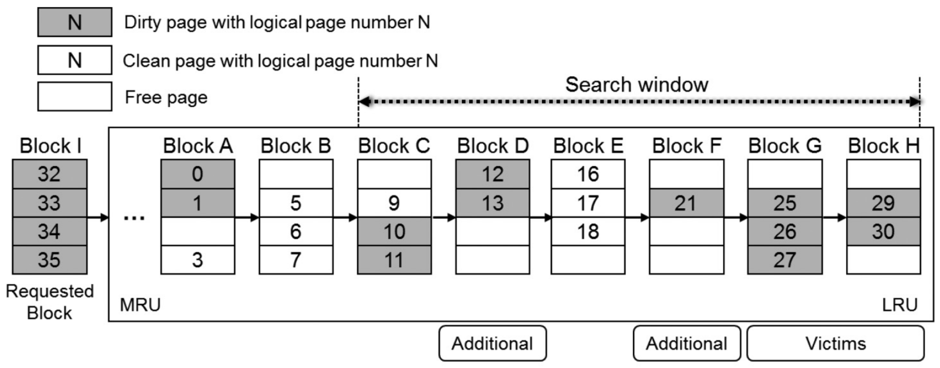
Fig. 8. Victim selection for read requests.

requests consist of two steps. At the first step, it secures buffer space for a requested block. After, it finds more victim blocks to be written in parallel.

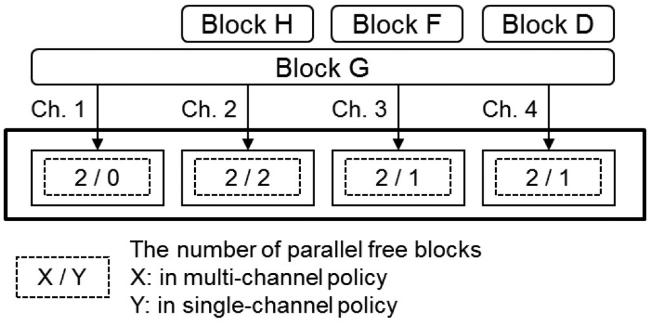
In the first step, the channel status is obtained from the pattern handler, that is, the number of parallel free blocks and garbage collection status. Then victims are selected from the LRU position until there are enough pages for the incoming block. Here, since clean victim blocks do not need an extra write operation, clean blocks are unconditionally selected. Dirty blocks are selected if it can be written in parallel with other victim blocks. If a block is not located at any channel or its all pages are re-written, then it takes a free block of any channel, since a new location in the NAND flash memory should be determined. If not, then it takes a free block of the currently located channel to prevent page migration among channels.

After the first step, additional victims can be selected. To avoid evicting a block near the MRU position that can be accessed with higher probability than a block near the LRU position, we set the search window. This window starts at LRU position and ends up at a location whose accumulative buffer hit counts from the MRU position reaches to a specific portion of entire buffer hit counts; in other words, the specific portion of buffer hits occur outside of the window. With this window, even if all blocks in the window are evicted, most of buffer hits are still available with blocks that are located outside of the window. We used 90% as the portion that the window size is large enough to select multiple dirty victims with keeping a similar level of the buffer hit ratio in our evaluations. Here, the end location of the window is re-calculated whenever an amount of requests comes into the buffer. Within the window, additional dirty victims are selected if they can be written in parallel with other victim blocks, in the same manner as the first step. However, clean blocks are not selected as additional victims to avoid decrement of the read hit ratio. After the eviction, as in the read request case, dirty pages and the pattern information of the victim blocks are sent to the pattern handler. Here, if flash memories are not busy, actual writing operations in the NAND flash area will be processed in the background. If they are busy for previous writing operations or GC, then waiting time can be occurred.

Figure 9 shows an example of the victim selection for write request, with 4 channels, 4 pages in a block, and a threshold value of 2. In Figure 9(a), at first, blocks H and G were selected as victims to provide enough space for block I. As shown in Figure 9(b), block H, which is a random block, will take a free block of channel 2 and block G, which is a sequential block, will take a free block of the multi-channel policy. After that, one more sequential block can be allocated to the multi-channel policy and two more random blocks can be allocated to channels 3 and 4 of the single-channel policy, so the buffer will find the additional dirty victim blocks in the search window. In that figure,



(a) Victim block selection


Block H
Block F
Block D
Block G
Ch. 1
Ch. 2
Ch. 3
Ch. 4
2 / 0
2 / 2
2 / 1
2 / 1
X / Y
The number of parallel free blocks
X: in multi-channel policy
Y: in single-channel policy

(b) Channel allocation

Fig. 9. Victim selection for write requests.

blocks F and D were selected as additional victim blocks and they will be allocated to channels 3 and 4. Here, block E was not selected as an additional victim, since it is a clean block.

Among the previous works, CFLRU [32] also manages clean and dirty pages in a single buffer, which targets the buffer cache of host file system or swap system. It uses page-level LRU, and clean pages will always be evicted first in the window. On the other hand, the SUPA's buffer is designed for internal buffer of an SSD with consideration of the pattern-based management on the FTL. It can select both clean and dirty victim blocks depending on the request type, due to properties of read and write operations of the NAND flash memories. In this process, we applied the new victim selection methods. For read request, clean blocks will be evicted first, but for write request, based on the preceding described algorithm, it selects multiple clean and dirty victim blocks based on channel status and pattern for better parallel access and pattern-based management.

Additionally, a buffer of Han et al. [10] uses hierarchical DRAM and non-volatile memory buffer to handle clean and dirty data. From DRAM buffer, dirty data are first moved into non-volatile memory for reliability. In SUPA, since we consider DRAM-only buffers, we concentrated on selecting clean victim blocks more than dirty blocks for performance improvement.

5.4 Pattern Handler and FTL

5.4.1 Basic Operation of FTL. Each request from the buffer is handled by the pattern handler before it goes to the FTL. The pattern handler monitors the statuses of channels in terms of garbage

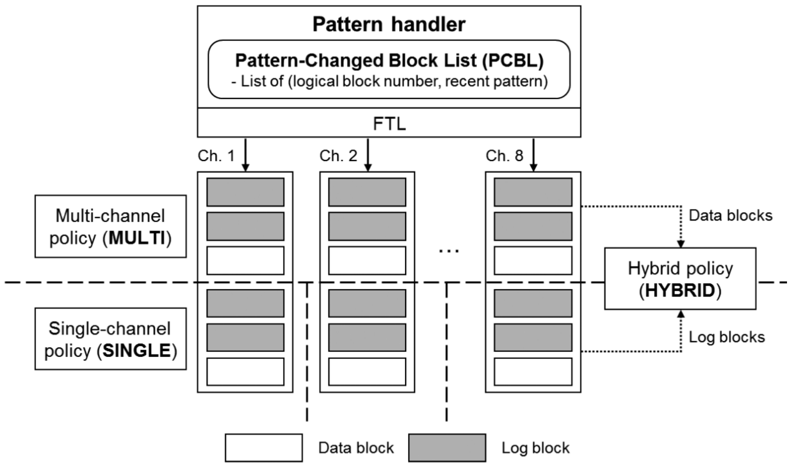

Pattern handler
Pattern-Changed Block List (PCBL)
- List of (logical block number, recent pattern)
FTL
Ch. 1
Ch. 2
Ch. 8
Multi-channel policy (MULTI)
Single-channel policy (SINGLE)
Data blocks
Hybrid policy (HYBRID)
Log blocks
Data block
Log block

Fig. 10. An architecture of the pattern handler and the FTL.

collections and the number of free blocks. It also allocates blocks to the appropriate channels and banks of the FTL depending on the victim selection results from the buffer. Each FTL policy has an amount of data blocks and log blocks. Data blocks are basic physical locations of logical blocks with block-level mapping. Updated pages are written in log blocks with page-level mapping. In the GC process, valid pages of data blocks and log blocks will be moved to new blocks, and those blocks will become new data blocks.

Figure 10 shows the architecture of the pattern handler and the FTL. Here, we use two different FTL policies: a multi-channel policy, MULTI, and a single-channel policy, SINGLE. MULTI is configured for full-channel page-level striping, while SINGLE is configured for a single channel per block. We use FTL policies of MNK [38], but other FTL policy can be adopted if it is based on a hybrid mapping policy that consists of data blocks and log blocks. The pattern handler manages free block allocations and garbage collections on both policies, since arbitrary management in each policy can interfere in each other. Additionally, there exists a hybrid policy, HYBRID. This policy consists of data blocks in MULTI and log blocks in SINGLE. Here, HYBRID does not use an independent NAND area, but just takes some parts of MULTI or SINGLE, to avoid additional policy switching overheads from HYBRID to one of MULTI and SINGLE, or vice versa. For pattern-changed blocks, updated pages will be written in the new policy, while other pages are remaining in the old policy. Here, valid pages can be found either MULTI or SINGLE, but not both. Blocks stored in HYBRID stay there if their patterns change repeatedly. If their patterns do not change any longer, then they will be located at the NAND area of corresponding policy.

5.4.2 Pattern Change Detection and Handling. The pattern handler detects pattern changes depending on the policy decision table as shown in Table 2, where MULTI, SINGLE, and HYBRID are the multi-channel policy, the single-channel policy, and the hybrid policy, respectively, and SW, RW, SR, and RR are sequential write, random write, sequential read, and random read patterns, respectively. Here, HYBRID stores pattern-changed blocks. For an incoming block, the pattern handler refers to its current policy and the incoming pattern, and a block will be identified as a pattern-changed block if policy switching should occur.

A block that has not been previously written is denoted as "None" in the table, and it will follow a policy of its write pattern except RW-SR. For RW-SR, since there is a policy collision between MULTI for read-intensive and SINGLE for random write, the pattern handler identifies it as

Table 2. FTL Policy Decision Table

Current policy	Incoming pattern	New policy	Pattern changed
None	SW, SW-SR, SW-RR	MULTI	No
	RW, RW-RR	SINGLE	No
	RW-SR	HYBRID	Yes
MULTI	SW, SR, RR, SW-SR, SW-RR	MULTI	No
	RW, RW-SR, RW-RR	HYBRID	Yes
SINGLE	RW, RR, RW-RR	SINGLE	No
	SW, SR, SW-SR, SW-RR, RW-SR	HYBRID	Yes
HYBRID	Any pattern	HYBRID	Yes

MULTI, multi-channel policy; SINGLE, single-channel policy;
HYBRID, data blocks in MULTI and log blocks in SINGLE;
SW, Sequential Write; RW, Random Write;
SR, Sequential Read; RR, Random Read.

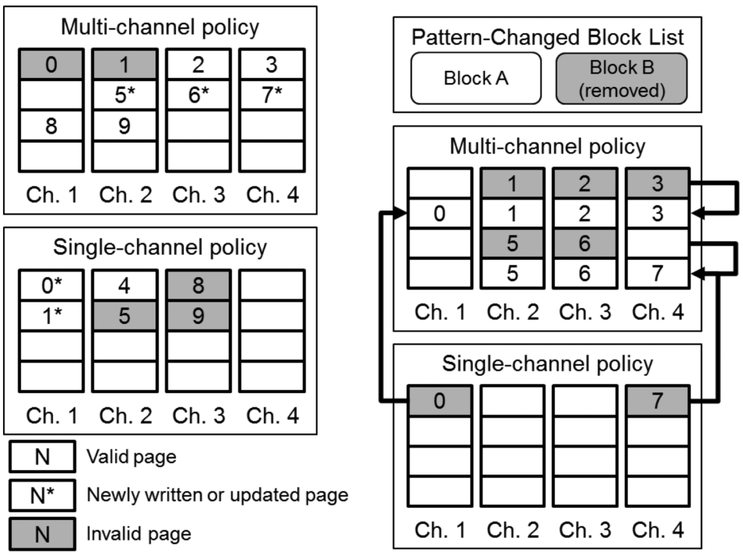
pattern-changed block. When a block is previously written in MULTI, if the incoming write pattern is random, it is identified as pattern-changed. For a block previously written in SINGLE, if the incoming write pattern is sequential or it is a read-intensive block, it is identified as pattern-changed. If a block is already located at HYBRID, then regardless of the incoming pattern, it remains as a pattern-changed block.

Blocks without pattern change are stored in MULTI or SINGLE as decided in victim selection process. To provide parallel access to multiple channels, sequential blocks and read-intensive blocks will be stored in MULTI. To reduce GC overhead from small size updates, random write-intensive blocks will be stored in SINGLE. If a pattern change is detected for a block, then the pattern handler inserts a logical block number (LBN) and the incoming pattern of that block in the pattern-changed block list (PCBL). While LBNs are located in PCBL, corresponding blocks use data blocks in MULTI and log blocks in SINGLE.

In the GC process, the pattern handler checks whether there are more pattern changes on each block or not. For a block in the PCBL, if the policy of recent two patterns is consistently either MULTI or SINGLE, this block will follow the corresponding policy, and it will be removed from the PCBL. If not, then all of the valid pages will be moved to MULTI where their data block is located, without removal from the PCBL.

Figure 11 shows how pattern-changed blocks are managed in the four-channel architecture. Here, we assume that each block contains four pages and the threshold value dividing sequential and random is 2. Figure 11(a) shows pattern-changed block mapping policies. When random write (LPNs 0 and 1) occurred for pages in MULTI (LPNs from 0 to 3), only updated pages will be stored in SINGLE to reduce garbage collection overhead on MULTI. When sequential write (LPNs from 5 to 7) occurred for pages in SINGLE (LPNs from 4 to 5), only update pages will be stored in MULTI to increase parallel access and reduce rewrite of sequential block of BLAS. When read-intensive occurred for pages in SINGLE (LPNs 8 and 9), block information is added to pattern-changed block list, and the pattern handler waits until next eviction of that block. If next pattern of that block is write-only, then updated pages will be stored in the policy of write pattern. If next pattern contains read accesses, then accessed pages will be stored in MULTI. In this progress, we can reduce rewrite of BLAS and increase parallel access.

Figure 11(b) shows garbage collection policies for pattern-changed blocks. For block A (LPNs from 0 to 3), if there are still pattern changes, a new data block is located at MULTI and the block



(a) Pattern-changed block mapping (b) Garbage collection

Fig. 11. Pattern-changed block management policy.

A still remains in the pattern-changed block list. For block B (LPNs from 5 to 7), if patterns for the same policy come continuously, all pages of the block B will be moved to corresponding policy, and its information will be removed from the pattern-changed block list.

From the previous pattern-based managements, SUPA can improve the performance. Unlike S-FTL [30], SUPA uses page-level striping for sequential or read-intensive blocks, and parallel access can occur for even single block. Compare to BLAS [4], SUPA can reduce some rewrite operations. In SUPA, pages with sequential write pattern will be directly scattered in multiple channels, while those pages should be rewritten in BLAS. Additionally, due to this direct scattering, when the pattern of the corresponding block is changed to read-intensive, pages also do not need rewrite operations. Compare to pattern-based CAVE [31, 38], since SUPA scatters pages of read-intensive blocks while pattern-based CAVE only considers write pattern, more parallel accesses for read requests can be provided. When write pattern is changed between sequential and random, HYBRID can manage policy switching between SINGLE and MULTI.

In terms of the lifetime, pattern-based managements including SUPA increase erase counts and valid page copy counts coming from switching policies. In case of SUPA, since it has less extra writes of switching policies, its erase count and valid page copy count will be somewhat reduced, compared to other pattern-based managements. The wear-leveling, however, is not much affected, since it is closely related to FTL policies. Even though the total write counts are increased, selecting appropriate physical location will stabilize the wear-leveling.

6 EVALUATION

6.1 Experimental Environment

In this section, we present the effects of SUPA through experiments. We first compared SUPA simulator with FlashSim [18] and a real SSD device (Samsung's SSD 850 PRO) [34] to validate the results. We then identified the effects of changing parameters. After that, we compared SUPA with S-FTL [30], BLAS [4], and pattern-based CAVE [31, 38] in terms of latency and lifetime.

Table 3. Simulation Parameters

Simulation parameter	Value
SSD size	80GB
The number of channels	8
The number of ways	4/channel
Buffer size	64MB
Page size	4KB
Block size	256KB
Log block portion	5%
Read latency	60us/page
Write latency	800us/page
Erase latency	1.5ms/block
Threshold to divide sequential and random	4 pages
Re-calculation period of searching window	10,000 I/O requests

We implemented SUPA and comparison works in a trace-driven simulator [31, 38, 39] for evaluations. The simulation parameters are given in Table 3. The total SSD size was set to 80GB, and the parallel NAND flash memory modules were configured with eight-channel and four-way architecture [31, 38]. The size of each page was 4KB, and the size of each block was 256KB; a single block contained 64 pages. The log block portion was set to 0.05. The specifications of each NAND flash memory were set as Samsung's MLC NAND flash memory [33]. The times taken in the NAND flash memory for read, write, and erase were set to 60us, 800us, and 1.5ms, respectively. The threshold value to divide sequential and random blocks was set to four pages. For the victim selection process of SUPA, the search window size was re-calculated when 10,000 I/O requests had come into the buffer.

Several disk I/O workload traces were used for evaluations. OLTP and WEBENGINE traces were made available courtesy of Ken Bates from HP, Bruce McNutt from IBM, and the Storage Performance Council [41]. The OLTP trace was extracted from OLTP applications running at two large financial institutions, and the WEBENGINE trace from a web search engine. The DBT2 trace was extracted by running the DBT2 benchmark tool for 2h [40] on MySQL 14.14 with 20 connections, 100 warehouses, and 50 terminals per warehouse. The GENERAL, SYSMARK, WEB, and MOV traces were obtained from a Microsoft Windows-based laptop computer [22]. These show typical PC usages: GENERAL for a 5-day-long general PC usage including office works, web surfing, and installations; SYSMARK for SYSmark 2007 Preview benchmark; WEB for one-day web surfing; and MOV for sequential movie files writing. RANWRITE is a synthesized workload from the IOZone benchmark [26]. Its several random writes are combined with sizes from 1 to 16KB. Here, WEBENGINE, MOV, and RANWRITE were used for the evaluations of extreme cases. Details of those traces are presented in Table 4, where volume sizes are measured by the maximum logical address. Here, OLTP and DBT2 have larger volume sizes than the simulated SSD, so we assumed that overflowed addresses are mapped to never-accessed addresses.

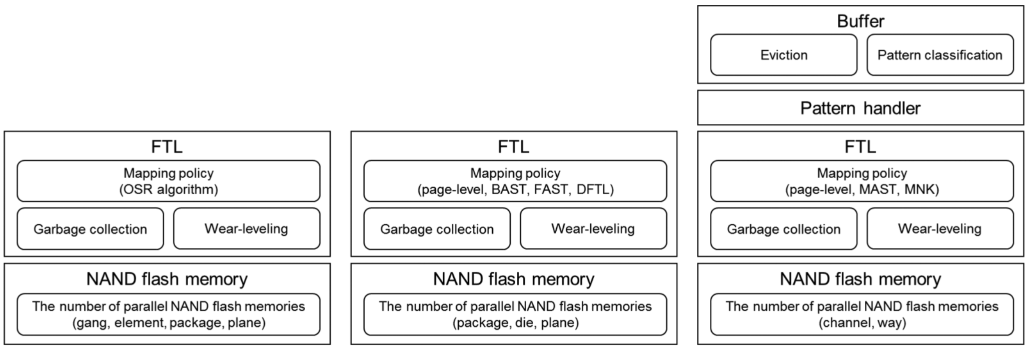
For performance metrics, we evaluated the read and write latencies and lifetime represented by erase counts, valid page copy counts, and wear-leveling of erase counts among channels. Using these metrics, we compared our work with other pattern-based managements.

6.2 Simulator Validation

Since 2011, we have been developing our own SSD simulator [31, 38, 39]. For SUPA, we added a pattern handler and modified buffer and FTL policies. We compared our results for a set of

Table 4. Workload Traces Specification

Workload	Request counts	Write ratio (%)	Total read / write size (GB)	Space utility (GB)	Volume size (GB)	Avg. read / write size (KB)
OLTP	5,334,987	76.83	4.45 / 20.61	0.54	644.53	3.78 / 5.27
DBT2	416,991	69.00	4.54 / 7.56	4.04	114.35	36.84 / 27.54
GENERAL	1,029,052	70.08	9.26 / 15.17	7.12	18.34	31.54 / 22.06
SYSMARK	150,092	62.39	2.65 / 4.10	5.39	31.94	49.16 / 45.87
WEB	200,605	76.92	0.94 / 3.57	2.17	16.83	21.37 / 24.25
WEBENGINE	4,579,809	0.02	65.82 / 0.01	6.46	16.67	15.07 / 8.10
MOV	430,024	98.86	0.29 / 26.74	7.85	8.00	62.69 / 65.95
RANWRITE	2,000,000	100.00	0.00 / 19.06	2.01	2.01	0.00 / 9.99


FTL
Mapping policy (OSR algorithm)
Garbage collection
Wear-leveling
NAND flash memory
The number of parallel NAND flash memories (gang, element, package, plane)
FTL
Mapping policy (page-level, BAST, FAST, DFTL)
Garbage collection
Wear-leveling
NAND flash memory
The number of parallel NAND flash memories (package, die, plane)
Buffer
Eviction
Pattern classification
Pattern handler
FTL
Mapping policy (page-level, MAST, MNK)
Garbage collection
Wear-leveling
NAND flash memory
The number of parallel NAND flash memories (channel, way)

(a) A structure of DiskSim with SSD extension (b) A structure of FlashSim (c) A structure of SUPA simulator

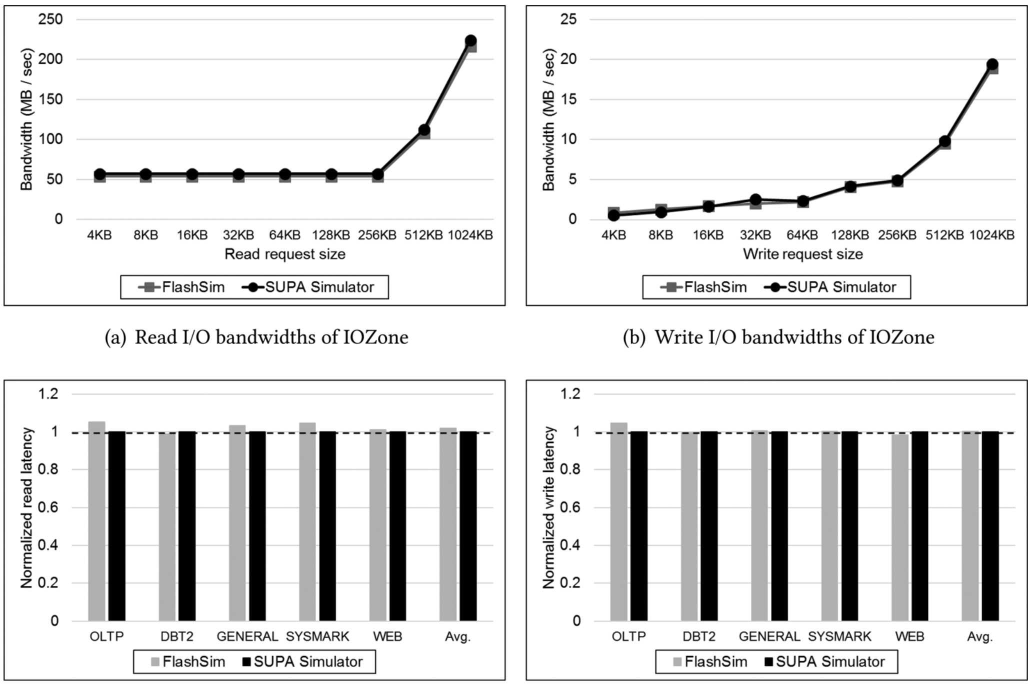
Fig. 12. Structures of SSD simulators.

workloads with the outputs using a well-known simulator and a real SSD device, to check how similar our simulator operates as other simulator and a real device.

DiskSim [3] with SSD extension [25] and FlashSim [2, 18] are well-known SSD simulators. Figures 12(a), 12(b), and 12(c) show the structures of DiskSim with SSD extension, FlashSim, and SUPA simulator, respectively. In DiskSim with SSD extension, the NAND flash memory layer supports multiple NAND flash memories for parallel access, which can be configured by the numbers of gangs, elements, packages, and planes. The FTL layer supports a policy called OSR, which operates page-level mapping policy. In FlashSim, parallel NAND flash memories are supported in terms of packages, dies, and planes. The FTL layer supports configurable mapping policy, garbage collection policy, and wear-leveling policy. Here, for mapping policies, the page-level mapping [1], BAST [16], FAST [21], and DFTL [9] are implemented on it.

SUPA simulator has similar NAND flash memory layer and FTL layer as other simulators. Multiple NAND flash memories can be configured by the numbers of channels and ways. For FTL management policies, the page-level mapping [1], MAST [39], and MNK [38] are implemented on it. In the SUPA simulator, however, there exist pattern handler and buffer layer on top of NAND flash memory and FTL layers. These layers simulate operations of pattern handler and buffer described in Section 5.

In the validation, we measured read and write performances of FlashSim and SUPA simulator in terms of bandwidth and latency. Since the FTL policies of SUPA are based on hybrid-mapping policy, we compare the SUPA simulator with FlashSim that basically includes several



(a) Read I/O bandwidths of IOZone

(b) Write I/O bandwidths of IOZone

(c) Read latencies of various workloads

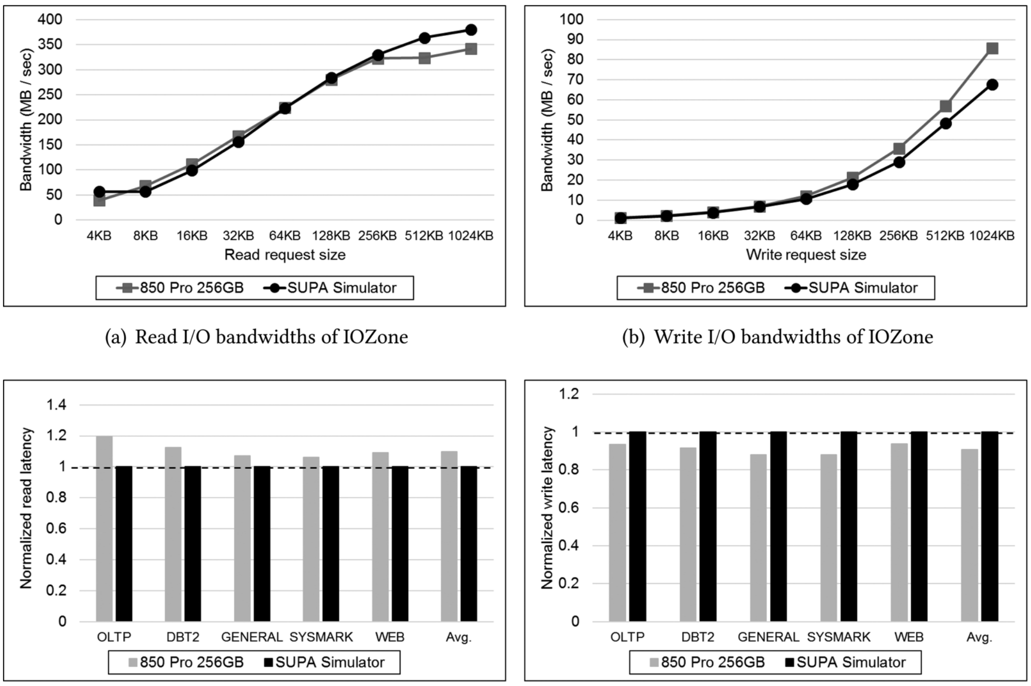
(d) Write latencies of various workloads

Fig. 13. Read and write bandwidths and latencies on simulators.

hybrid-mapping policies. In FlashSim, since there is no buffer layer and pattern handler layer, we compare the results from the FTL layer and NAND flash memory layer, bypassing the buffer layer and pattern handler layer of SUPA simulator. For the comparison of FTL and NAND flash memory layer, we configure the management policies of SUPA simulator like the policies implemented in FlashSim. For the FTL management policy, we simulated the same BAST of FlashSim on our SUPA simulator. In the NAND flash memory layer, we configure 32 NAND flash memories, with (package,die,plane)=(8,1,4) in FlashSim and (channel,way)=(8,4) in SUPA simulator.

At the first of validations, IOZone benchmark [26] is used to determine I/O bandwidths of two simulators with file size 10GB and I/O sizes from 4KB to 1024KB, with parameters in Table 3. Figures 13(a) and 13(b) show read and write I/O bandwidth of two simulators, respectively. For various I/O sizes, both read and write bandwidth have less than 3% of differences in two simulators. Here, some differences can be made by physical location selections for write requests. Additionally, two simulators show same read and write bandwidth trends. For read requests, bandwidths are just proportional to the number of accessed parallel channels. They show almost same read bandwidth until 256KB request size, which is the size of single block, and the bandwidth grows based on parallel access. Bandwidths for write requests have the same trend as the read case, but there exists bandwidth grow in small request sizes by the effect of garbage collections, since larger write request size can reduce valid page copying overhead.

We then compared their read and write latencies with workload traces described in Table 4. Figures 13(c) and 13(d) show normalized read and write latencies by results of SUPA simulator on two simulators, respectively. On average, they show read and write latencies with less than 2% differences.



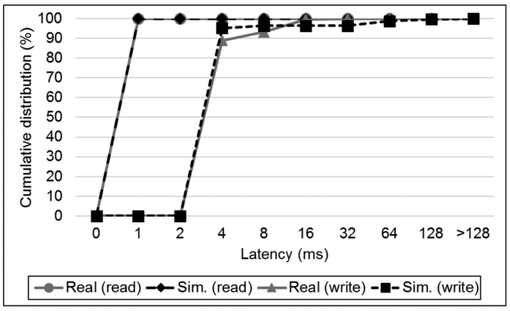
(a) Read I/O bandwidths of IOZone (b) Write I/O bandwidths of IOZone

(c) Read latencies of various workloads (d) Write latencies of various workloads

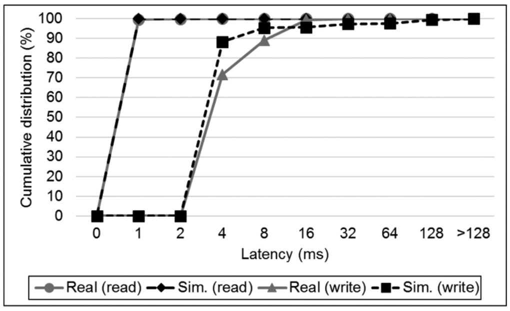
Fig. 14. Read and write bandwidths and latencies on a real SSD device.

We also compared the SUPA simulator with a real SSD device. Samsung's SSD 850 PRO is used with 256GB capacity, 512MB DRAM cache, and 550 and 520MB/s maximum sequential read and write performances, respectively [34]. It is connected to CentOS 7.2 host by SATA3, and ext4 file system is used. During the evaluation, a host-level buffer cache is bypassed. We compared the SUPA simulator with the real SSD device using the same manner as the comparison with Flash-Sim. For the SUPA simulator, we used 256GB capacity and a full-channel page-level striping FTL policy. We also added latencies from device connection interfaces such as the file system and data communication; in our observation, additional latencies for read and write requests are 0.1 and 2.5ms, respectively. Other parameters are the same as the comparison with FlashSim.

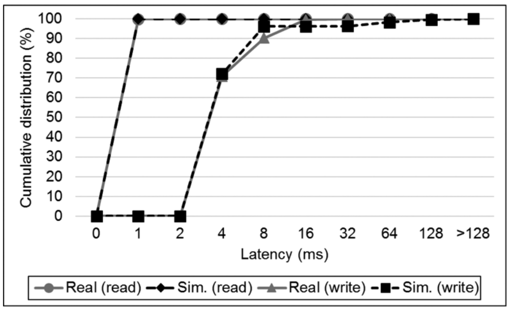
Figures 14(a) and 14(b) show read and write bandwidths, respectively, from IOZone benchmark. They show similar read bandwidths with 10.0% differences. For write bandwidths, the SUPA simulator shows 11.5% smaller bandwidth values than the real SSD device, with the same trend; bandwidths grow as a request size gets larger, but slower growth than read case. For large write sizes, the bandwidth differences can come from their internal architectures and algorithms, such as internal parallelism, garbage collection policy, and low-level hardware specifications. For small write sizes, since those differences may not affect performances that much, the real SSD and the SUPA simulator show almost same bandwidths. Figures 14(c) and 14(d) show their read and write latencies with workload traces described in Table 4, respectively. Their read and write latencies show similar differences through all workloads. On average, the SUPA simulator shows 9.8% smaller read latencies and 10.3% larger write latencies than the real SSD device, which are similar differences as shown in Figures 14(a) and 14(b).



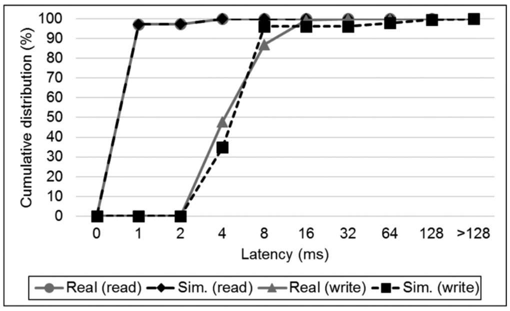
(a) OLTP



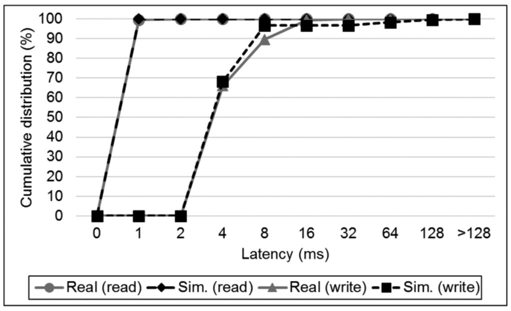
(b) DBT2



(c) GENERAL



(d) SYSMARK



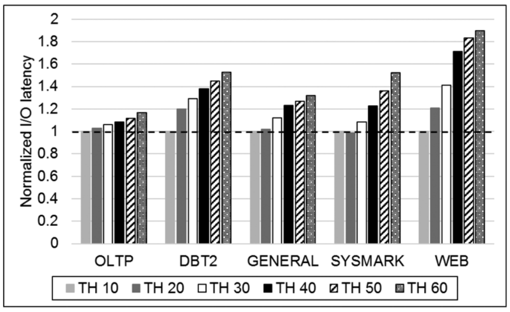
(e) WEB

Fig. 15. Cumulative distributions of read and write latencies for various workloads.

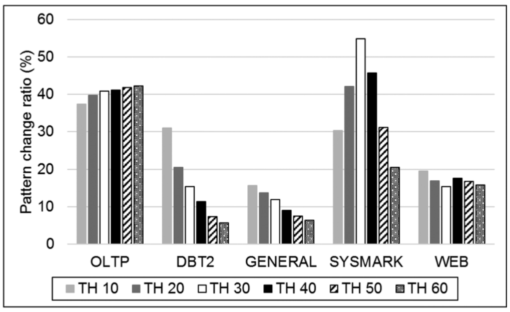
Figures 15(a), 15(b), 15(c), 15(d), and 15(e) show cumulative distributions of read and write latencies for OLTP, DBT2, GENERAL, SYSMARK, and WEB, respectively, on the real SSD device and the SUPA simulator. For all workloads, they show almost similar read distributions. For write distributions, they have some variations that can come from different internal architectures and algorithms, but they still show similar trends. With the preceding observations, therefore, we can validate the SUPA's simulator, and we used its results in the later evaluations.

6.3 Parameter Configuration

Before comparing SUPA with others, we analyzed how each parameter could affect SUPA. We evaluated the relationship between the pattern change ratio and latencies with various threshold values, and the buffer hit ratio of the SUPA's buffer with changing buffer sizes.

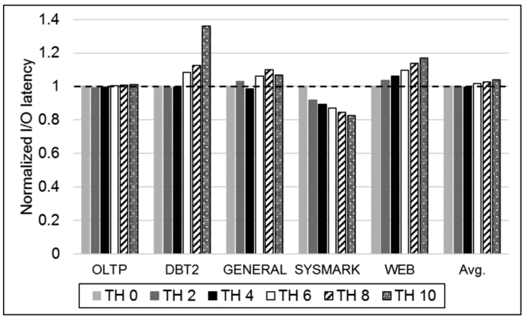


(a) Normalized latencies

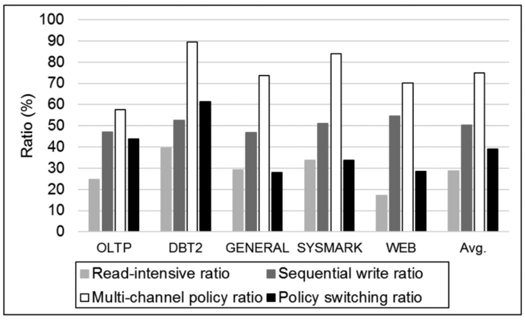


(b) Pattern change ratios

Fig. 16. Normalized latencies and pattern change ratios with various thresholds.



(a) Normalized latencies on small threshold values



(b) Pattern statistics with threshold value 4

Fig. 17. Normalized latencies and pattern statistics with small threshold values.

6.3.1 Pattern Distribution with Various Threshold Values. We first evaluated the pattern change ratio between sequential and random, and latency to see their relationship. Figure 16(a) shows I/O latencies normalized by the threshold value 10, and Figure 16(b) shows pattern change ratios of SUPA without the pattern handler. OLTP, GENERAL, and WEB showed relatively stable pattern change ratios, while DBT2 and SYSMARK's pattern change ratios changed dramatically. I/O latencies, however, varied regardless of the pattern change ratio, since handling some pattern-changed blocks with only MULTI or SINGLE can reduce the pattern change ratio, but can increase latencies; handling sequential blocks on SINGLE increases latency by the lack of parallel access, and handling random blocks on MULTI increases GC overhead. Therefore, we focused on reducing the pattern change overhead rather than reducing the pattern change ratio by selecting the appropriate threshold value.

To determine the threshold value for later experiments, we ran SUPA with small sizes of the threshold values. Figure 17(a) shows I/O latencies normalized by the threshold value 0. Each workload had different latency variation, and we selected the threshold value with smallest average latency, which was 4. Because the page size was 4KB, a block was classified as sequential if the size of accessed pages were more than 16KB. Figure 17(b) shows pattern statistics of victim blocks for each workload with threshold value 4. On average, 28.8% of victim blocks have read-intensive pattern and 50.4% of victim blocks have sequential write pattern. Additionally, 74.9% of victim blocks are lead to be stored in multi-channel policy, and policy switches are required for 39.1% of victim blocks.

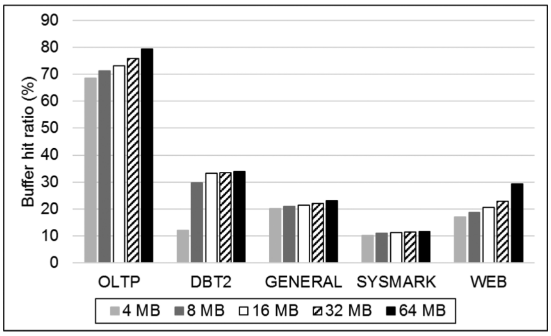


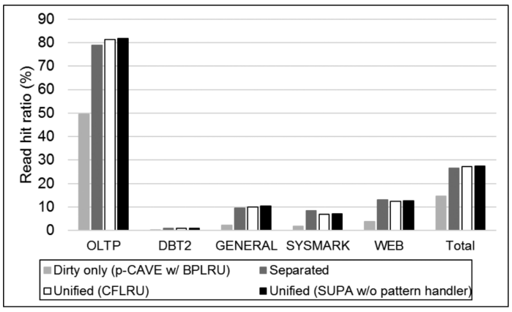
Fig. 18. Buffer hit ratios of SUPA with various buffer sizes.

6.3.2 Buffer Hit Ratio with Various Buffer Sizes. Next, we increased the buffer size from 4 to 64MB to determine the relationship between buffer size and hit ratio. Figure 18 shows the buffer hit ratios according to buffer sizes. There is no doubt that the buffer hit ratios are increased as the buffer sizes are increased. The increment factor of buffer hit ratio, however, was not as fast as that of buffer size. While the buffer size increased by a factor of 16 (4 to 64MB), buffer hit ratio increased by factors of only 1.16, 2.79, 1.14, 1.13, and 1.72 for OLTP, DBT2, GENERAL, SYSMARK, and WEB, respectively. This is because most buffer hits occurred near the MRU position with spatial and temporal localities [32], so multiple victim selection will not affect the buffer hit ratio that much. Additionally, although clean blocks take possession of the buffer without co-hits, it will not decrease the write hit ratio that much.

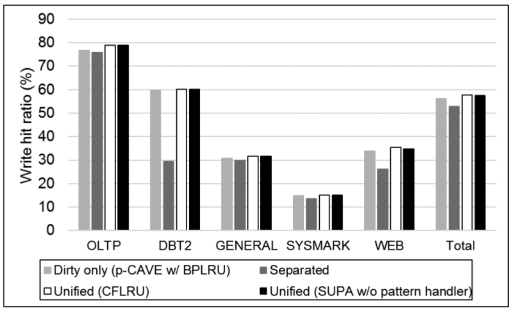
6.4 Performance Comparison

Now, we compared SUPA with other pattern-based managements in terms of buffer hit ratio, read and write latencies, and lifetime to verify performance improvement. We first evaluated the buffer hit ratio according to several buffer schemes. We then compared it with other pattern-based managements, S-FTL [30], BLAS [4], and pattern-based CAVE (p-CAVE) [31, 38], to check latencies and lifetime improvement from the buffer policy and the pattern change handling process. For the FTL policies, we used full-channel page-level striping for MULTI, and single channel per block without page-level striping for SINGLE.

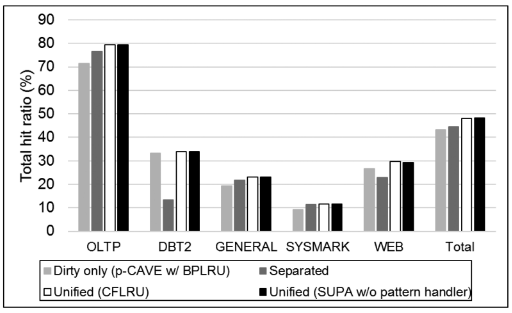
6.4.1 Buffer Hit Ratio and Latency. To determine the effectiveness of the SUPA's buffer, we evaluated its buffer hit ratio in comparison with a dirty-only buffer, which is the typical write buffer, a clean and dirty separated buffer, and unified buffers. Here, p-CAVE is used for a dirty-only buffer, whose eviction policy is BPLRU [15]. For unified buffers, CFLRU [32] and the SUPA's buffer are compared. Figures 19(a), 19(b), and 19(c) show the read, write, and total buffer hit ratios, respectively. In the Figure 19(a), the read hit ratios of dirty-only buffer are not 0, since reading data in the dirty-only buffer can also be returned from the buffer. The ratios increase with the buffer scheme changed as dirty-only, separated, and unified, except for DBT2. For DBT2, the dirty-only buffer has almost similar hit ratio as unified buffers, because read hit ratios are less than 1% for all buffers. For the separated buffer, using a separated linked list can increase the read hit ratio as much as the unified buffer, but the write hit ratio is smaller, since it cannot determine the exact LRU order between clean and dirty blocks. In total, the SUPA's buffer showed a 5.0%p and 3.8%p higher buffer hit ratio than the dirty only and separated buffer, respectively. Two unified buffers show almost same buffer hit ratios; the difference is less than 0.1%p.



(a) Read hit ratios

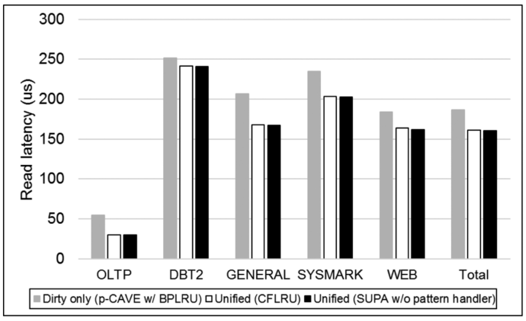


(b) Write hit ratios

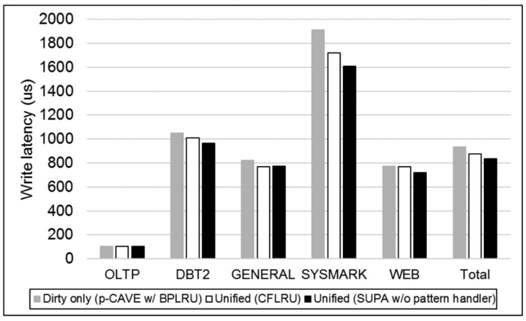


(c) Total hit ratios

Fig. 19. Buffer hit ratios of various buffers.



(a) Read latencies



(b) Write latencies

Fig. 20. Read and write latencies for various buffers.

Although two unified buffers have similar buffer hit ratios, however, CFLRU shows longer latencies than the SUPA's buffer. Figures 20(a) and 20(b) show the read and write latencies of buffers, respectively. CFLRU and the SUPA's buffer show almost the same read latencies, which are reduced by 16.2% from dirty-only buffer, since clean pages and blocks can exist in the buffer. In case of write latency, although CFLRU reduces write latencies by 7.0% from dirty-only buffer due to co-hit, it shows 4.8% longer write latencies than the SUPA's buffer because of the absence of consideration in patterns and multi-channel parallelism. With these observations, we can conclude that the SUPA's buffer can increase buffer hit ratios as CFLRU, and it can reduce write latencies more than CFLRU.

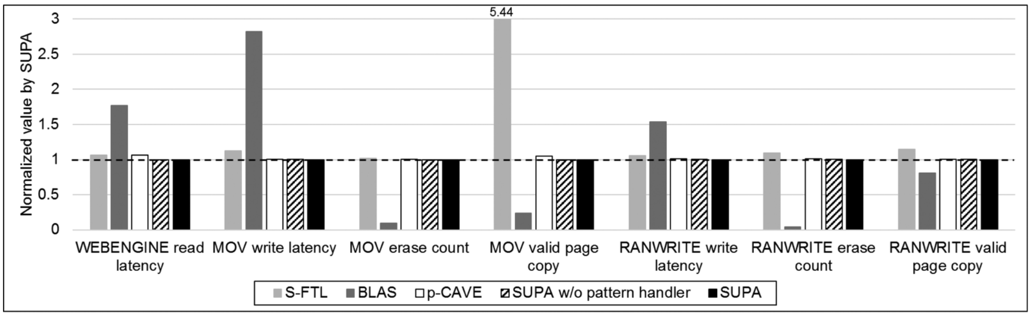


Fig. 21. Latencies and lifetimes for read-intensive (WEBENGINE), sequential write-intensive (MOV), and random write-intensive (RANWRITE) workloads.

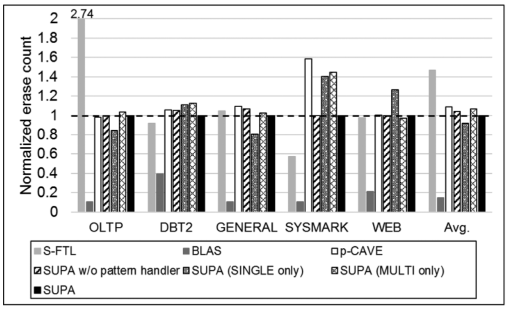
6.4.2 Performance and Lifetime without Pattern Changes. We ran WEBENGINE, MOV, and RANWRITE as extreme cases, since they do not have any pattern change. WEBENGINE is a read-intensive workload with only a few write requests, MOV is a sequential write-intensive workload, and RANWRITE is a random write-intensive workload.

Figure 21 shows the read latency of WEBENGINE and the write latencies, erase counts, and valid page copy counts of MOV and RANWRITE. For BLAS, since both read-intensive and sequential write-intensive blocks should be rewritten after their first write within NAND flash memory, its latencies are larger than the others. On SUPA, a new buffer policy decreased the read latency for the WEBENGINE workload by 6.6%, since the unified buffer can manage clean blocks. For the MOV workload, SUPA can reduce 12.2% of write latency than S-FTL, because the buffer for random blocks of S-FTL cannot handle any block. When we compared the write latencies of pattern-based CAVE, SUPA without pattern handler and SUPA, there was almost no difference. For lifetime, BLAS has the least erase count and valid page copy count, since it uses page-level mapping only. SUPA has as same lifetime as other works. Similarly, for the RANWRITE workload, SUPA can reduce 14.6% of write latency than S-FTL. In the case of BLAS, although the workload is random write-intensive, since patterns of victim blocks can be changed to sequential while they stay in the buffer, some rewrites for sequential blocks will be needed. Through these observations, we can conclude that the additional management overhead incurred by SUPA does not affect the total performance and the lifetime.

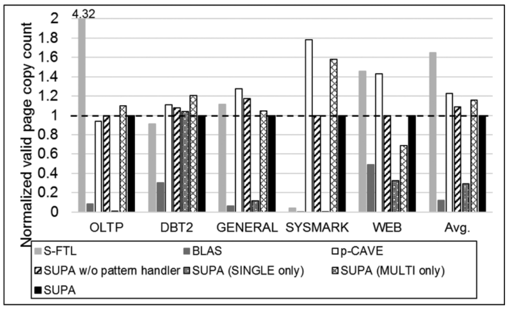
6.4.3 Lifetime. Next, we compared lifetime in terms of erase count, valid page copy count, and wear-leveling. Figures 22(a) and 22(b) show erase count and valid page copy count normalized by SUPA, respectively. Here, SUPA with SINGLE only stores all blocks in SINGLE with the threshold value 64, and SUPA with MULTI only stores all blocks in MULTI with the threshold value 0. The values of S-FTL changed dynamically depending on the workloads due to the absence of page-level striping. BLAS has smallest erase and valid page copy counts, since it uses a page-level mapping policy only. SUPA showed almost the same erase count and valid page copy count as other works, which means our additional schemes do not affect to the lifetime of an SSD.

To confirm that our schemes do not affect wear-leveling, we evaluated distribution of erase counts of channels. For wear-leveling evaluation, we calculated a coefficient of variation (CV), which is the ratio of the standard deviation to the mean value, also known as relative standard deviation, as Equation (1):

$$\textit{Coefficient of Variation (CV)} = \frac{\textit{standard\_deviation}}{\textit{mean}}. \tag{1}$$



(a) Erase counts



(b) Valid page copy counts

Fig. 22. Lifetimes of pattern-based managements.

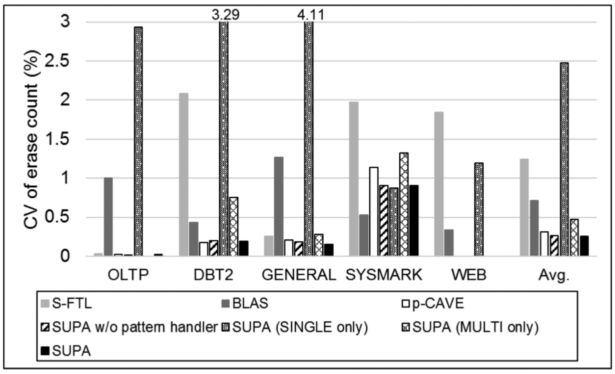
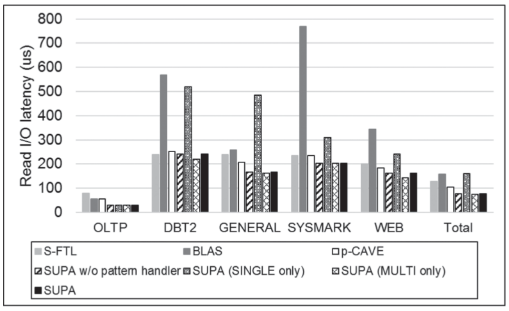


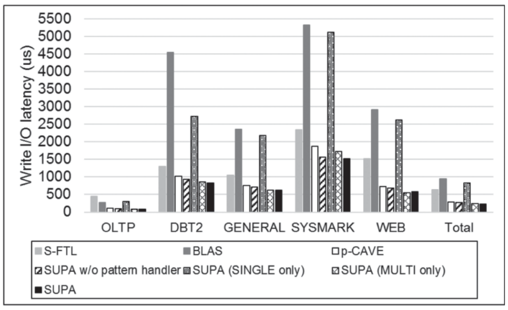
Fig. 23. Coefficient of Variation (CV) values of erase counts of channels.

If the CV is small, then each element is closer to the mean value. Figure 23 shows the CV values of erase counts of channels as a percentage. Although lazy scattering on BLAS and frequent small size rewrites on S-FTL can increase the erase count, they show nice wear-leveling with the CV values less than 2.5%. Pattern-based CAVE showed better wear-leveling with CV value 0.31%, since it scatters pages of sequential blocks at the eviction and selects multiple victims for multiple channels for random blocks. SUPA showed CV value 0.25%, so our channel selection progress does not incur a loss of wear-leveling.

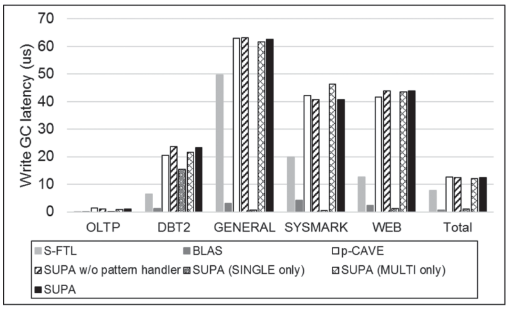
6.4.4 Overall Performance. Now, we compared overall performances in terms of read latency and write latency to show that SUPA has improvement with the new buffer policy and the pattern change handling. Here, we evaluated spent time on the buffer, the pattern handler, the FTL, and the NAND flash memory, and we excluded waiting time for previous requests. Figure 24(a) shows the read I/O latency. In cases of read requests, there is no time taken for garbage collection. SUPA with MULTI only shows best read latency, since each request is always done with parallel channels. In total, SUPA can get 1.6, 2.0, and 1.3 times smaller read latency than S-FTL, BLAS, and p-CAVE, respectively. Here, the pattern handler was not effective to reduce read latency that much, and almost all reduction came from the buffer management scheme, since other pattern-based managements are based on dirty-only buffers. Compared to p-CAVE, 33.6% of improvement came from the buffer management, and remaining 1.0% of improvement came from the pattern handler.



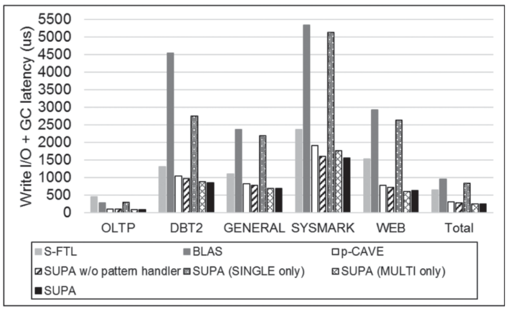
(a) Read I/O latencies



(b) Write I/O latencies



(c) Write garbage collection (GC) latencies



(d) Write I/O + garbage collection (GC) latencies

Fig. 24. I/O and garbage collection latencies of pattern-based managements.

Figures 24(b), 24(c), and 24(d) show the write I/O latency, garbage collection latency, and I/O + garbage collection latency, respectively. The trend of write I/O latency is similar as read case. For garbage collection latency, BLAS has smallest garbage collection overhead due to page-level mapping, and not much improvement on SUPA. Here, actual I/O latency is much larger than garbage collection latency, since the garbage collection is occurred only for some write requests, but not all of them. In total, SUPA can get 2.6, 3.9, and 1.3 times smaller write I/O + garbage collection latency than S-FTL, BLAS, and p-CAVE, respectively. Compared to p-CAVE, 32.8% of improvement came from the buffer management, and the remaining 67.2% of improvement came from the pattern handler.

7 CONCLUSION

In this article, we proposed a new buffer and FTL management scheme called SUPA, a single unified read-write buffer and pattern-change-aware FTL for high performance of multi-channel SSD. Because read and write requests have locality, which is closely related to the performance, we handle clean and dirty blocks in a unified read-write buffer. Additionally, we adopt different eviction policies for read and write requests based on read and write property of the NAND flash memory and multi-channel parallelism. To reduce the pattern change handling overhead caused by an old policy or policy switching, we use a pattern handler between the buffer and the FTL. The pattern handler detects pattern changes for evicted blocks from the buffer. With the pattern handler, pattern-changed blocks are stored according to a hybrid mapping policy of the previous and new policies. Our evaluation showed that, in comparison to previous pattern-based managements, SUPA can get up to 2.0 and 3.9 times smaller read and write latency, respectively, while maintaining the same lifetime.

REFERENCES

[1] A. Ban. 1995. U.S. patent US08027131: Flash file system.

[2] Matias Bjørling. 2011. Extended FlashSim. Retrieved from https://github.com/MatiasBjorling/flashsim.

[3] John Bucy, Jiri Schindler, Steve Schlosser, and Greg Ganger. 2008. The disksim simulation environment - version 4.0 reference manual. In *Carnegie Mellon University Technical Report*. Retrieved from http://www.pdl.cmu.edu/DiskSim/.

[4] Da-Wei Chang, Hsin-Hung Chen, Dau-Jieu Yang, and Hsung-Pin Chang. 2014. BLAS: Block-level adaptive striping for solid-state drives. *ACM Trans. Design Auto. Electron. Syst. (TODAES)* 19, 2 (2014), 21.

[5] Li-Pin Chang. 2008. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'08)*. IEEE, 428–433.

[6] Li-Pin Chang and Chun-Da Du. 2009. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Design Auto. Electron. Syst. (TODAES)* 15, 1 (2009), 6.

[7] Li-Pin Chang, Yi-Hsun Huang, and Chen-Yi Wen. 2011. On the management of multichannel architectures of solid-state disks. In *Proceedings of the 2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'11)*. IEEE, 37–45.

[8] Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 507–512.

[9] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, 229–240. DOI: http://dx.doi.org/10.1145/1508244.1508271

[10] S. J. Han, D. H. Kang, and Y. I. Eom. 2016. Hybrid write buffer algorithm for improving performance and endurance of nand flash storages. In *Proceedings of the 2016 IEEE International Conference on Consumer Electronics (ICCE'16)*. 83–84. DOI: http://dx.doi.org/10.1109/ICCE.2016.7430530

[11] Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, and Zheng Zhang. 2014. Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks. *ACM Trans. Archit. Code Optim.* 11, 4, Article 43 (Dec. 2014), 19 pages. DOI: http://dx.doi.org/10.1145/2677160

[12] Heeseung Jo, Jeong Uk Kang, Seon Yeong Park, Jin Soo Kim, and Joonwon Lee. 2006. FAB: Flash-aware buffer management policy for portable media players. *IEEE Trans. Consum. Electron.* 52, 2 (2006), 485–493.

[13] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. ACM, 161–170.

[14] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. 2009. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.* 58, 6 (2009), 744–758.

[15] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the Conference on File and Storage Technologies (FAST'08)*, Vol. 8. 1–14.

[16] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. Consum. Electron.* 48, 2 (2002), 366–375.

[17] Jin Hyuk Kim, Sang Hyuk Jung, and Yong Ho Song. 2008. Cost and performance analysis of nand mapping algorithms in shared-bus multi-chip configuration. In *Proceedings of the 3rd International Workshop on Software Support for Portable Storage (IWSSPS'08)*, Vol. 3. 33–39.

[18] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the 2009 First International Conference on Advances in System Simulation (SIMUL'09)*. IEEE Computer Society, Washington, DC, 125–131. DOI: http://dx.doi.org/10.1109/SIMUL.2009.17

[19] Se Jin Kwon and Tae-Sun Chung. 2015. Data pattern aware FTL for SLC+MLC hybrid SSD. *Des. Autom. Embedded Syst.* 19, 1–2 (March 2015), 101–127. DOI: http://dx.doi.org/10.1007/s10617-014-9138-5

[20] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operat. Syst. Rev.* 42, 6 (2008), 36–42.

[21] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst. (TECS)* 6, 3 (2007), 18.

[22] Yong-Goo Lee, Dawoon Jung, Dongwon Kang, and Jin-Soo Kim. 2008. μ-FTL: A memory-efficient flash translation layer supporting multiple mapping granularities. In *Proceedings of the 8th ACM International Conference on Embedded Software*. ACM, 21–30.

[23] Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. 2010. FASTer FTL for enterprise-class flash memory SSDs. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI'10)*. IEEE, 3–12.

[24] Dongzhe Ma, Jianhua Feng, and Guoliang Li. 2011. LazyFTL: A page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 1–12.

[25] Microsoft. 2009. SSD Extension for DiskSim Simulation Environment. Retrieved from https://www.microsoft.com/en-us/download/details.aspx?id=52332.

[26] William D. Norcott and Don Capps. 2003. IOZone filesystem benchmark. Retrieved from http://www.iozone.org.

[27] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst. (TECS)* 7, 4 (2008), 38.

[28] Dongchul Park, Biplob Debnath, and David H. C. Du. 2011. A workload-aware adaptive hybrid flash translation layer with an efficient caching strategy. In *Proceedings of the 2011 IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'11)*. IEEE, 248–255.

[29] JungWook Park, Gi-Ho Park, Charles Weems, and ShinDug Kim. 2009. Sub-grouped superblock management for high-performance flash storages. *IEICE Electron. Express* 6, 6 (2009), 297–303.

[30] Seung-Ho Park, Jung-Wook Park, Shin-Dug Kim, and Charles C. Weems. 2012. A pattern adaptive NAND flash memory storage structure. *IEEE Trans. Comput.* 61, 1 (2012), 134–138.

[31] Sung Kyu Park, Youngwoo Park, Gyudong Shim, and Kyu Ho Park. 2011. CAVE: Channel-aware buffer management scheme for solid state disk. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 346–353.

[32] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and JoonWon Lee. 2006. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, NY, 234–241. DOI:http://dx.doi.org/10.1145/1176760.1176789

[33] SAMSUNG. 2006. Datasheet: K9XXG08UXM (NAND Flash Memory. Retrieved from http://downloads.qi-hardware.com/hardware/datasheets/qi_lb60/U9~K9GAG08U0M~~2GB-FINAL_NAND-NOT_ON_SOME_PROTOTYPES~~.pdf.

[34] SAMSUNG. 2015. Datasheet: Samsung SSD 850 PRO. Retrieved from http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_850_PRO_Data_Sheet_rev_2_0.pdf.

[35] Dongyoung Seo and Dongkun Shin. 2008. Recently-evicted-first buffer replacement policy for flash storage devices. *IEEE Trans. Consum. Electron.* 54, 3 (2008), 1228–1235.

[36] Jinho Seol, Hyotaek Shim, Jaegeuk Kim, and Seungryoul Maeng. 2009. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 137–146.

[37] Yoon Jae Seong, Eyee Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin-yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, Yookun Cho, and Sang Lyul Min. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 7 (2010), 905–921.

[38] Gyudong Shim, Sung Kyu Park, and Kyu Ho Park. 2012. MNK: Configurable hybrid flash translation layer for multi-channel SSD. In *Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering (CSE'12)*. IEEE, 445–452.

[39] Gyudong Shim, Youngwoo Park, and Kyu Ho Park. 2011. A hybrid flash translation layer with adaptive merge for SSDs. *ACM Trans. Stor. (TOS)* 6, 4 (2011), 15.

[40] MySQL Benchmark Tool. 2002. DBT2 Benchmark Tool. Retrieved from https://dev.mysql.com/downloads/benchmarks.html.

[41] Trace Repository UMass. 2007. OLTP Application and Search Engine I/O. Retrieved from http://traces.cs.umass.edu/index.php/Storage/Storage.

[42] Chundong Wang and Weng-Fai Wong. 2012. ADAPT: Efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. IEEE, 1–12.

[43] Jia-Hao Wang, Hsin-Hung Chen, Wei-Jian Su, and Da-Wei Chang. 2011. Cross-layer optimizations in solid-state drives. *IEEE Embed. Syst. Lett.* 3, 4 (2011), 109–112.

[44] Qingsong Wei, Cheng Chen, and Jun Yang. 2014. CBM: A cooperative buffer management for SSD. In *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. IEEE, 1–12.

[45] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. 2011. WAFTL: A workload adaptive flash translation layer with data partition. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE, 1–12.

[46] Chin-Hsien Wu and Tei-Wei Kuo. 2006. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06)*. IEEE, 601–606.

[47] Guanying Wu, Ben Eckart, and Xubin He. 2010. BPAC: An adaptive write buffer management scheme for flash-based solid state drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–6.
[48] Wei Xie, Yong Chen, and Philip C. Roth. 2016. ASA-FTL: An adaptive separation aware flash translation layer for solid state drives. *Parallel Comput.* (2016).
[49] Wei Xie, Yong Chen, and Philip C. Roth. 2016. Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives. In *Proceedings of the 2016 IEEE International Conference on Networking, Architecture and Storage (NAS'16)*. IEEE, 1–10.

Received June 2016; revised January 2017; accepted July 2017