Des Autom Embed Syst (2010) 14: 415–443
DOI 10.1007/s10617-010-9066-y

Co-optimization of buffer layer and FTL in high-performance flash-based storage systems

Hyotaek Shim · Dawoon Jung · Jaegeuk Kim · Jin-Soo Kim · Seungryoul Maeng

Received: 21 September 2009 / Accepted: 4 October 2010 / Published online: 4 November 2010
© Springer Science+Business Media, LLC 2010

Abstract NAND flash-based storage devices have rapidly improved their position in the secondary storage market ranging from mobile embedded systems to personal computer and enterprise storage systems. Recently, the most important issue of NAND flash-based storage systems is the performance of random writes as well as sequential writes, which strongly depends on their two main software layers: a Buffer Management Layer (BML) and a Flash Translation Layer (FTL). The primary goal of our study is to highly improve the overall performance of NAND flash-based storage systems by exploiting the cooperation between those two layers. In this paper, we propose an FTL-aware BML policy called *Selective Block Padding* and a BML-based FTL algorithm called *Optimized Switch Merge*, which overcome the limitations of existing approaches on performance enhancement. When using both the proposed techniques, evaluation results show that the throughput is significantly increased over that of previous studies.

H. Shim (✉) · S. Maeng
Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), 335 Gwahangno, Yuseong-gu, Daejeon, Republic of Korea
e-mail: htshim@calab.kaist.ac.kr

S. Maeng
e-mail: maeng@calab.kaist.ac.kr

D. Jung · J. Kim
Memory Division, Samsung Electronics Co., Ltd., Hwaseong, Republic of Korea

D. Jung
e-mail: dw0904.jung@samsung.com

J. Kim
e-mail: jaegeuk.kim@samsung.com

J.-S. Kim
School of Information and Communication Engineering, Sungkyunkwan University (SKKU), 300 Cheoncheon-dong, Jangan-gu, Suwon, Republic of Korea
e-mail: jinsookim@skku.edu

Keywords Buffer management layer · Flash translation layer · NAND flash memory · Solid state drive

1 Introduction

As a notable non-volatile storage medium, NAND flash memory [1, 2] has been widely adopted in mobile embedded devices, for instance, mobile phones, digital cameras, and MP3 players, because it supports many attractive features such as low power consumption, low access latency, and shock resistance [3–5]. In addition, the capacity of NAND flash memory is under continuous development, extending its coverage from small mobile storage systems to large-scale server storage systems [6].

Recently, NAND flash-based Solid State Drives (SSDs) have been introduced, which offer the same I/O interface as that of Hard Disk Drives (HDDs) [7, 8]. SSDs provide better random access performance than HDDs by removing mechanical operations including disk platter rotation and disk arm movement. A multi-channel architecture [9] for SSDs has also enhanced sequential access performance beyond HDDs. As a competitive alternative to HDDs, SSDs are extending their share in the secondary storage market [10]. The advent of SSDs has been accelerating the migration from HDDs to NAND flash-based storage devices.

However, NAND flash-based storage devices, simply called *flash storage devices*, have a drawback where the random write latency is significantly longer than the random read latency due to some idiosyncrasies of NAND flash memory. Basically, a write (or program) operation takes longer than a read operation, and data must be erased in bulk before being overwritten in NAND flash memory. These features make SSDs considerably vulnerable to random writes [10]. Accordingly, when we employ SSDs in personal computer and server storage systems with complicated write patterns, achieving high performance is a challenging issue [11].

To hide the awkward features of NAND flash memory, flash storage devices dispose a software layer called a *Flash Translation Layer* (FTL) that carries out logical-to-physical address mapping, providing block device interface. The random write latency of flash storage devices strongly relies on FTL algorithms including address mapping algorithms, mapping granularity, and flash erase policies.

Flash storage devices also adopt a small-sized DRAM as a buffer cache, which is faster than NAND flash memory, to absorb random writes and to serve frequently requested data instead of FTL [11–16]. The buffer cache is controlled by a software layer called a *Buffer Management Layer* (BML), which is another key factor for performance enhancement. BML policies, such as how to select and evict victims, determine input request patterns for FTL and thus have a significant impact on the FTL behavior. Therefore, to obtain opportunities for extraordinary performance improvement, BML and FTL should be taken into account together.

In recent years, to enhance the performance of flash storage systems, many studies have focused on developing BML policies and FTL algorithms, separately. Previous approaches can be classified as follows. First, various FTL algorithms have been devised by adjusting trade-offs between performance gains versus memory consumption for storing mapping information. This approach is very effective for tuning flash storage systems to the characteristic of workloads. However, most existing FTL schemes [17–25] assume that there is no information about BML policies or there is no buffer cache. Because write patterns are entirely changed through BML before reaching FTL, it is necessary to develop FTL algorithms that consider the BML policies. Second, several flash-aware buffer management schemes

have been proposed [11–14]. These schemes exploit the superficial behaviors of FTL algorithms and the characteristics of NAND flash memory. In this approach, BML strives to generate write patterns favorable for reducing the overhead of FTL. However, those existing schemes assume traditional FTL algorithms optimized for write patterns from the host, not from BML, missing opportunities for better performance.

To overcome the limitation of the existing studies that improved either the BML policy or the FTL algorithm, each should be aware of and optimized for the other. In this paper, we focus on the cooperative optimization between both the software layers. This advanced optimization is possible in common flash storage devices, such as SSDs and various mobile embedded devices, where those two software layers can be executed on the same embedded processor [15].

As a co-optimization (CO-OP) scheme, we introduce two novel techniques for BML and FTL, respectively. One is an FTL-aware BML policy, called *Selective Block Padding* (SBP), that provides FTL with well-suited write patterns by recognizing the state information of FTL. The other is a BML-assisted FTL algorithm, called *Optimized Switch Merge* (OSM), that significantly reduces extra operations in NAND flash memory, supported by the SBP technique. For this CO-OP scheme, we devised additional interfaces between BML and FTL, which will be explained in Sect. 5. Through trace-driven simulation, we confirm that the proposed scheme achieves the best performance under various workloads and evaluation configurations, compared with the previous studies.

The rest of this paper is organized as follows. Section 2 contains brief descriptions about the architecture of typical flash storage systems, NAND flash memory, and FTL algorithms. Section 3 explains the previous studies on flash-aware buffer management and buffer cache-aware FTL. Section 4 gives an explanation of the effect of a block padding technique. Section 5 describes the proposed techniques in detail. Section 6 explains simulation environment and analyzes evaluation results. Finally, Sect. 7 summarizes and concludes this paper.

2 Backgrounds

2.1 NAND flash-based storage system

Figure 1 shows the overall architecture of a typical NAND flash-based storage system including two main software layers: BML and FTL. When a read request arrives in the flash storage device, BML checks whether the requested data exist in the DRAM-based buffer cache. If the data is there, the read request is served by the buffer cache, otherwise is redirected to FTL. On a write request, if the write data is previously cached, the data is simply overwritten. For a new write request, BML allocates empty buffer space. If there is no more empty space, BML selects and flushes victim buffer data to FTL. Considering the longer write latency and erase operations in NAND flash memory, it is more effective to use the buffer cache only for write caching, not for read caching, in flash storage devices [11, 15].

2.2 Characteristics of NAND flash memory

NAND flash memory consists of an array of blocks, each of which contains a fixed number of pages, and it offers three basic operations: read, write (or program), and erase operations. The unit of read and write operations is a page, and the unit of erase operations is a block. A read operation obtains data from a page, while a write operation stores data into a page. An erase operation clears all data in a block. There are two types of NAND flash memory.

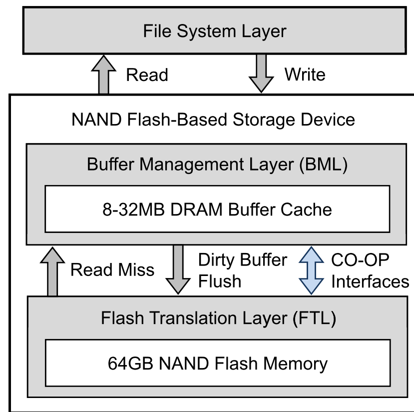


Fig. 1 Architecture of a NAND flash-based storage system

Table 1 Specification of NAND flash memory (Samsung Electronics K9WAG08U1M [1], K9GAG08UXM [2])

Flash type	Unit size (KB)		Access time (μs)		
	Page	Block	Read	Write	Erase
SLC	2	128	72.8	252.8	1500
MLC	4	512	165.6	905.6	1500

Single Level Cell (SLC) NAND [1] stores one bit per cell, whereas Multi Level Cell (MLC) NAND [2] stores two or more bits per cell, supporting larger capacity. The specific operation time and unit size are shown in Table 1.

As previously mentioned, NAND flash memory has some awkward characteristics. It suffers from asymmetric read/write latency, which means that write operations take quite longer than read operations. In addition, there is the *erase-before-write* characteristic that write operations can be allowed only on previously-erased pages. Since the unit of erase operations is larger than the unit of write operations, before erasing a block, we should copy valid pages in the block to another previously-erased block, which is called *valid page migration* or *valid page copies*. When using *large block NAND* flash memory, there is an additional restriction where all pages within a block must be written in sequence from the first to the last page.

In NAND flash memory, the number of erase operations on a block is limited, typically, from 5,000 to 100,000 [1, 2]. If the erase count of a block is over the limit, the block is likely to be worn out and cannot be written any more. Hence, erase operations should be evenly distributed in all blocks, which is called *wear leveling*. Considering this constraint, we need to reduce the number of erase operations not only to enhance the performance but also to extend the lifetime of flash storage devices.

2.3 Power failure recovery

Buffered data and mapping information stored in a volatile DRAM-based cache can be lost by unexpected power failures. Simple approaches to prevent the loss of the important data in the device cache are to employ either (1) non-volatile memory devices [26] such as phase change RAM (PRAM) [27] and ferroelectric RAM (FRAM) [28], (2) traditional battery-backed DRAMs, or (3) *supercapacitor* that provides enough power to flush all of the dirty data in the device cache to NAND flash memory.

Without the non-volatile buffer cache, the host can periodically issue flush cache commands that completely flush the write cache to alleviate the loss of write-cached data. This will be helpful for reducing the possibility of data loss, but the write cache may be almost empty, increasing write operations in NAND flash memory.

The *Lightweight Time-shift Flash Translation Layer* (LTFTL) is an example of software-based approach that aims at maintaining FTL consistency in case of abnormal shutdown [29]. In this scheme, FTL maintains previous data pages in the log area without reclaiming them until a checkpoint. After detecting the abnormal shutdown at initialization, LTFTL turns back to the previous state of time periods, which preserves consistency, by changing the mapping of data pages in the log area to their previous data pages. This technique is well suited for the *out-of-place update* property of NAND flash memory.

2.4 Flash Translation Layer

The main role of FTL is to make a flash storage device emulate a traditional storage device that provides a block device interface, covering the idiosyncrasies of NAND flash memory; through this compatibility, flash storage devices can be easily adopted in existing storage systems. To cope with the erase-before-write characteristic, FTL assigns write requests to empty pages, which are erased in advance, and updates the logical-to-physical mapping information. In this way, the pages that contain old data are invalidated. If there are no available empty pages, FTL selects victim blocks and triggers *garbage collection* in order to recycle them as free blocks. Before reclaiming the victim blocks, FTL carries out valid page migration from the victim blocks to some free blocks reserved for garbage collection. After this, the victim blocks are converted to free blocks by erase operations.

According to mapping granularity, we divide existing FTL schemes into three categories: page mapping, block mapping, and hybrid mapping schemes. In page mapping schemes, a logical page number is translated to a physical page number in NAND flash memory. Due to the flexibility in assigning empty pages, the best performance is accomplished even for random write patterns [17]. However, they demand large memory resources to maintain the fine-grained mapping information. Moreover, the size of mapping information considerably increases in proportion to the capacity of flash storage devices. For example, when assigning 4 bytes for each page mapping entry with MLC NAND flash memory shown in Table 1, we need a 64 MB memory for 64 GB flash capacity. Accordingly, the page mapping scheme is not suitable for consumer-based flash storage devices that involve the constraint of memory resources. To alleviate such a problem, the *Demand-based FTL scheme* (*DFTL*) [18] has been developed to create balance between the performance and the memory consumption. Basically, DFTL applies a caching mechanism to existing page mapping schemes.

In block mapping schemes, FTL maintains coarse-grained mapping information that translates a logical block number to a physical block number in NAND flash memory. Among three categories, block mapping schemes require the smallest memory resources to store the mapping information, while their performance is quite low due to the restriction that all the pages in a block must be fully and sequentially written and located on their appointed offset. Accordingly, to overwrite some pages in a block, the other valid pages in the block should be migrated to a free block together with the new pages. This valid page migration seriously degrades the performance of flash storage systems.

To balance the advantages of page and block mapping schemes, hybrid mapping schemes have been devised. Basically, this scheme is similar with the block mapping scheme since it offers block mapping for most of the blocks called *data blocks*, but it also supports page mapping for a small fixed number of blocks called *log blocks* to handle write requests.

Incoming write data is written in the log blocks incrementally from the first page, thus reducing the overhead of valid page migration. When all free blocks are consumed, FTL copies all valid pages within victim log blocks and their related data blocks into reserved free blocks, which should be fully and sequentially written to become new data blocks. Thereafter, FTL erases the victim log blocks and old data blocks.

Many hybrid mapping schemes have been developed [19–23]. Among them, *Block Associative Sector Translation* (*BAST*) [19] and *Fully Associative Sector Translation* (FAST) [20] are well known for competitive performance with the small usage of computing and memory resources, and they are most widely used in research and industrial areas [11, 13–15, 25, 30]. In this paper, we focus on developing the co-optimization techniques in these two famous hybrid mapping schemes.

2.4.1 Block-Associative Sector Translation (BAST)

The BAST scheme (or the *log block scheme*) allocates at most one log block for each data block. Write requests to a data block can be written in its allocated log block until all empty log pages are consumed, when FTL merges the log block with the corresponding data block. Figure 2 illustrates three types of merge operations in BAST: a switch merge, partial merge, and full merge.

As shown in Fig. 2(a), if all pages in a log block are sequentially written in their designated page offset, FTL can simply convert the log block into a new data block, and the original data block is reclaimed by an erase operation, which is called a *switch merge*. Figure 2(b) illustrates a *partial merge*. If a log block is partially written in a sequential manner from the first page, FTL fills up the remaining empty pages of the log block by copying the omitted pages from the data block, and then it operates like a switch merge. Figure 2(c) describes a *full merge*. If a log block is written in non-sequential order, FTL must copy the valid pages owned by the log block and the data block to another free block, which becomes a new data block. Then FTL erases the log and old data blocks, and converts them into free blocks. This full merge requires N page reads, N page writes, and two block erases, where N is the number of pages per block.

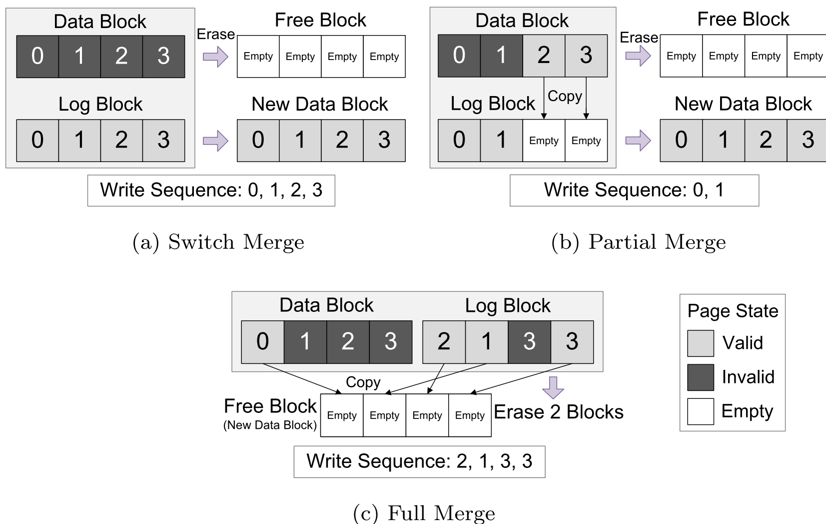


Fig. 2 Three types of merge operations in BAST

In addition to the merge operations caused by fully-written log blocks, when FTL falls short of free blocks, one of the pre-allocated log blocks should be reclaimed, e.g., in Least Recently Used (LRU) order. If the number of log blocks is smaller than the working set size of write patterns, FTL can suffer *log block thrashing*, which means that log blocks allocated for data blocks are repeatedly reclaimed before consuming all of their empty pages, lowering the utilization of the log blocks.

2.4.2 Fully-Associative Sector Translation (FAST)

To prevent log block thrashing, *Fully Associative Sector Translation* (FAST) [20] has been proposed. In this scheme, there are two types of log blocks, RW and SW log blocks. FAST allocates only one SW log block for sequential writes, and the remaining log blocks are used for handling random writes as RW log blocks. In FAST, all random updates for data blocks can be located in any RW log blocks, since all data blocks share all RW log blocks. From this, FAST brings two advantages. First, RW log blocks can be fully utilized. Second, log pages in RW log blocks are likely to be invalidated by overwriting, reducing overheads for valid page migration. Those invalidated pages in RW blocks can increase more if there are enough RW log blocks and the workload exhibits high temporal locality for write requests.

If there are no more free RW log blocks, one of them is reclaimed by a round-robin fashion. To reclaim an RW log block, FAST should merge all associated data blocks that keep valid pages in the RW log block. To merge an associated data block, FAST copies all valid pages that belong to the associated data block into a free block by searching all RW log blocks. After this, the free block becomes a new data block and the associated data block is erased. This is called a *full merge*. These full merges are repeated until all the associated data block are merged into new data blocks, and finally the victim RW log block is erased.

FAST distinguishes random and sequential writes as follows. If the first-page offset of a write request is zero (the first page in a block) and the SW log block is empty, or if the first offset is located on the next page of the recently-written page in the SW log block, the write request is considered as sequential writes and is written into the SW log block. In this way, the SW log block can be written only in a sequential manner from the first to the last page. If the SW log block is fully written, it becomes a new data block, and the old data block becomes a free block simply by a switch merge. If the first-page offset of a write request is zero but the SW log block is not empty, FTL triggers a partial merge for the SW log block to accommodate the write request. For the other cases, write requests are stored into RW log blocks, considered as random writes.

3 Related works

The *Flash-Aware Buffer* (FAB) management scheme [13] was proposed to reduce the merge cost of log block-based FTLs by increasing the probability of switch merges. In FAB, buffer pages that belong to the same logical block are managed together by a block-level LRU policy. If the buffer cache becomes full, FAB selects a buffer block that has the largest number of valid pages as a victim block and then flushes all dirty pages in the victim block. If there are multiple buffer blocks that have the largest number, block-level LRU order is applied to select a victim block among them. This *largest block-based victim selection* exploits a behavioral characteristic of log block-based FTLs; when complete blocks are flushed to FTL, namely *complete-block flushes*, switch merges are likely to occur. Accordingly, FAB helps FTL change costly full merges into low-cost switch merges, and provides better performance under sequential write patterns, compared with a page-level buffer replacement policy.

Block Padding Least Recently Used (BPLRU) [11] is another buffer management scheme optimized for log block-based FTLs to cope with even random write patterns. This scheme consists of two main techniques: *LRU compensation* and a *page padding technique*, which we call a block padding technique in this paper. In BPLRU, a victim block in the buffer cache is selected by block-level LRU order.

First, LRU compensation improves the effectiveness of the block-level LRU policy in the buffer cache. If a buffer block is fully written, the block is moved to the LRU position, not the Most Recently Used (MRU) position. This technique assumes that a fully-written buffer block is unlikely to be overwritten shortly, considering sequential write patterns. LRU compensation is effective for addressing the mixed patterns of sequential and random writes.

Second, when a victim buffer block is flushed, the block padding technique fills up all omitted pages in the victim block by reading them from FTL, and flushes the complete block to FTL. This technique eliminates full and partial merges, and always derives switch merges from FTL. That is why the performance of BPLRU is not affected by the number of log blocks in FTL. Under sequential write patterns, this scheme is highly effective to diminish the cost of merge operations with few log blocks.

To reduce the padding overheads, the *Block Padding Recently-Evicted-First* (*BP-REF*) scheme [14] devised a conditional block padding technique. This scheme keeps the fixed number of victim buffer blocks that contain the largest number of pages within a victim window. If the number of dirty pages in the victim block is larger than a predefined threshold, the victim block is flushed after being padded. Otherwise, BP-REF flushes an LRU page within the victim blocks. This conditional padding technique can be more cost-effective than the unconditional padding technique of BPLRU under random write patterns. However, the conditional padding technique can cause numerous full merges involving padding overheads if padded victim blocks are repeatedly flushed to partly-written log blocks in BAST.

The *Buffer-Aware Garbage Collection* (BA-GC) scheme [15] was also proposed to reduce the merge cost of FTL. In BA-GC, unlike the previous studies that concentrated on the BML policies, a buffer-aware FTL technique was introduced. This scheme targets the environment where FTL can access and control the buffer cache. During garbage collection, some read operations for valid page copies are served from the buffer cache instead of NAND flash memory if the valid pages are cached in the buffer cache, which is called *Buffer-Aware Block Merge* (BA-BM). Thereafter, the states of the accessed dirty buffer pages are changed to *clean*, since these buffer pages already hold the same data as those in NAND flash memory, thus avoiding following flushes. To amplify the effect of BA-BM, the number of dirty buffer pages for each log block is taken into consideration when a victim log block is selected in FTL, which is called *Buffer-Aware Victim Block Selection* (BA-VBS).

In BA-GC, if many dirty buffer pages are rewritten shortly after their states are changed to clean by BA-BM, the benefit that reduces dirty page flushes will diminish. To avoid this scenario, BA-GC uses the *3-region LRU buffer* that maintains an update probability for each buffer block, which is used to sort out *cold pages* that are unlikely to be updated. Supported by the 3-region LRU buffer, BA-GC selects log blocks that have many dirty and cold buffer pages as victim log blocks in FTL to decrease the number of dirty pages to be flushed and read operations for valid page migration.

4 Motivation

The previous studies explained in the above section have been developed to improve the performance of flash storage systems. One of them, BPLRU, is a well-known scheme that

Table 2 Model parameters to analyze the trade-offs of the unconditional block padding technique

Notation	Description
N	# of pages per block
N_f	# of buffer flushes corresponding to the target log block
N_i	# of dirty pages of the ith buffer flush
N_σ	# of dirty pages flushed to the target log block during N_f flushes ($N_\sigma = \sum_{i=1}^{N_f} N_i$)
N_r (Sect. 5.1.3)	# of remaining empty pages in the target log block
T_r, T_w, T_e	Read, write, erase operation time in NAND flash memory
GC_{FM}	*Full merge* cost: $N \cdot (T_r + T_w) + 2 \cdot T_e$
GC_{SM}	*Switch merge* cost: T_e

focused on elevating the random write performance through refining buffer management policies. In this scheme, BML is optimized for FTL, while the FTL algorithm is unaware of the BML policies. In other words, BML struggles to make beneficial input requests for FTL without cooperation with FTL, thus giving severe restriction to enhancing the performance.

To illustrate this limitation, we analyze the effect of the block padding technique proposed in BPLRU [11], which provides trade-offs between removing costly full merges and increasing the overhead caused by unconditionally padding a victim block for each flush. If a victim block is almost full of valid pages, the performance benefit of replacing full merges with switch merges exceeds the padding overhead. Conversely, if a victim block has few valid pages, the padding overhead can be considerably larger than the benefit.

With respect to the trade-offs, we compare the block padding technique with the non-padding technique. To find parameters that determine the costs of both the techniques, we set up a model, where we use the notations summarized in Table 2. This model calculates the cost occurred on one log block in FTL as follows. Within a given period, there are N_f flushes of a victim buffer block that corresponds to the target log block simply called the log block. For each flush, a victim buffer block contains N_i $(0 < N_i \leq N)$ dirty pages, whose sum is N_σ that means the total number of flushed dirty pages during N_f flushes. N_f ranges from 1 up to N_σ, and the case of $N_f = N_\sigma$ is even possible, for example, if a victim block is flushed N_σ times with only one dirty page.

We define C_{pad} as the cost including the padding overhead during the given period with the block padding technique. During the same period, we also define C_{nopad} as the cost without the block padding technique. In this analysis, we consider only full merges for C_{nopad}, assuming that the log block is written in non-sequential order, with the consideration of random write patterns; in cases where switch merges are made by sequential flush requests, the non-padding technique is more cost-beneficial. If N_σ is not a multiple of N, the log block is partly written after the last flush in C_{nopad}. In this case, we assume that the log block is ultimately reclaimed by a full merge to make the same state of the log block for both the techniques after the last flush. The costs of the two techniques are calculated by the following formulas:

$$C_{nopad} = \sum_{i=1}^{N_f} N_i \cdot T_w + \left\lceil \frac{\sum_{i=1}^{N_f} N_i}{N} \right\rceil \cdot GC_{FM}$$

$$= N_\sigma \cdot T_w + \left\lceil \frac{N_\sigma}{N} \right\rceil \cdot GC_{FM}$$

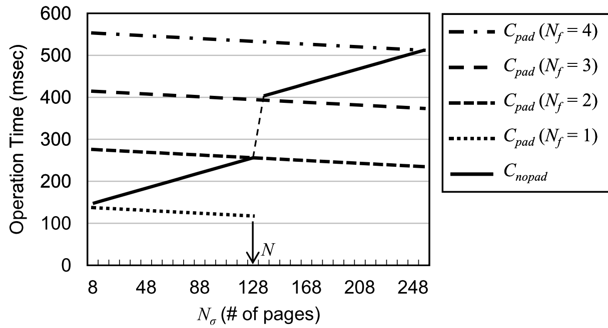


Fig. 3 Cost comparison of the padding and non-padding techniques

$$C_{pad} = \sum_{i=1}^{N_f} \big(N_i \cdot T_w + (N - N_i) \cdot (T_r + T_w) + GC_{SM}\big)$$
$$= (N_f \cdot N - N_\sigma) \cdot T_r + N_f \cdot N \cdot T_w + N_f \cdot GC_{SM}$$

C_{nopad} involves N_i page writes for each flush and overall $\lceil N_\sigma / N \rceil$ full merges including the log block redemption after the last flush. C_{pad} contains N_i page writes, the padding overhead ($N - N_i$ page reads and writes), and one switch merge, for each flush. Figure 3 shows a comparison of C_{pad} and C_{nopad} according to N_f and N_σ. To calculate the operation time, we used the specification of MLC NAND flash memory shown in Table 1. Note that if $N_f = 1$, N_σ is limited from 1 to N, since the number of dirty pages of a victim block in the buffer cache cannot exceed N.

As N_σ increases C_{nopad} increases, which is regardless of N_f, but C_{pad} is seriously affected by N_f rather than N_σ. In C_{pad}, N_σ / N_f means the average rate of dirty pages for all flushes, which we simply call the *dirty rate*. As N_f decreases under the same N_σ, the dirty rate increases and the padding overhead is mitigated, achieving better performance for the block padding technique (C_{pad}). However, the larger the number of flushes (N_f), the larger C_{pad} beyond C_{nopad} because each of flushes accompanies considerable padding overheads for victim blocks with the small dirty rate. Under random write patterns, the dirty rate is likely to be lower than that under sequential write patterns, incurring a great number of flash reads and writes for padding. Moreover, considering that N_f ranges up to N_σ in the worst case, the padding overhead may overwhelm the benefit of removing full merges.

This analysis describes the limitation of the block padding technique when using the unmodified FTL algorithm that is unaware of the BML policies. This technique exploits only superficial information of FTL behavior; this is about the condition to generate switch merges in FTL. Accordingly, this technique unconditionally pads a victim block, thus resulting in numerous extra operations. Therefore, we need a more intelligent BML policy as cross-layer optimization through communicating with FTL in order to ensure better and stable performance not affected by the dirty rate.

5 Co-optimization between BML and FTL

For better performance of flash storage systems, BML and FTL should be aware of and optimized for each other. From this point of view, the BML policy can cooperate with the FTL algorithm with the consideration of the FTL state to provide appointed input requests, and FTL should be prepared for the refined input requests. As a co-optimization (CO-OP)

Table 3 CO-OP interfaces provided by FTL for BML

Notation	Description		
BAST	# of free pages in the LBN log block/Ø	←	Get_NFreePG_Log(LBN)
	True/false	←	Is_Seq_Log(LBN)
	# of free log blocks	←	Get_NFreeLog()
	LBN of the next victim log block	←	Get_Victim_Log()
FAST	Threshold length for random write requests	←	Get_Thres_RW()

scheme, we propose two techniques: *Selective Block Padding* (SBP) and *Optimized Switch Merge* (OSM). We apply these techniques to two popular FTL algorithms: BAST and FAST. These hybrid FTLs are widely adopted in various cost-sensitive consumer-based flash storage devices due to their small memory and computing resource consumption. For this co-optimization, we state the CO-OP interfaces from BML to FTL, which will be used in the following SBP algorithms, for the two target FTLs in Table 3.

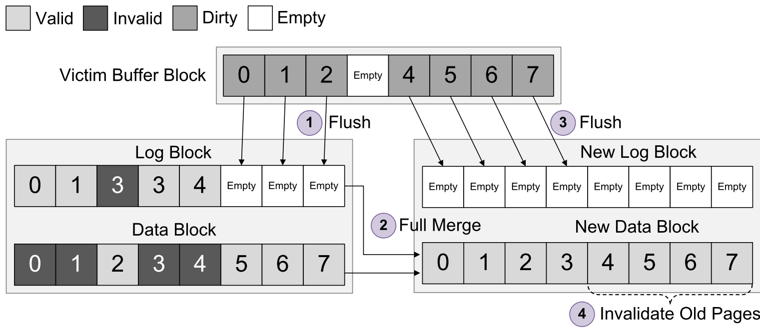
5.1 Case study with Block-Associative Sector Translation (BAST)

In the CO-OP scheme with BAST, a victim block in the buffer cache is flushed to FTL with padding only if a full merge is expected through the flush, and then FTL exploits this *complete-block flush* to avoid the full merge. By means of this cooperative optimization, we propose an advanced flushing process, whose one example is presented and compared with a non-optimized case in Fig. 4. Figure 4(a) illustrates a flushing process without the co-optimization, and it proceeds as follows. The remaining empty pages of the log block are written by the front dirty pages (0, 1, and 2) of the victim buffer block, and the fully-consumed log block is reclaimed by an expensive full merge. Thereafter, a log block is reallocated, and the remaining dirty pages (4, 5, 6, and 7) are written into the new log block. In this example, the total cost of the flushing process is 8 page reads, 15 page writes, and 2 block erases in NAND flash memory.

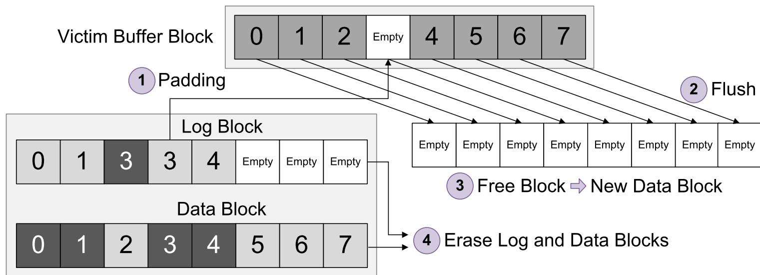
In the advanced flushing process shown in Fig. 4(b), the CO-OP scheme pads the victim block before flushing it in anticipation of a full merge that will be triggered by the flush, and then the victim block is fully written to another free block instead of the original log block. After being written, the free block becomes a new data block, and then the old log and data blocks are erased. In this way, CO-OP saves 7 page reads and writes, compared with the flushing process shown in Fig. 4(a). Moreover, following merge operations can be further delayed since FTL obtains a completely-empty log block after the buffer flush. When adopting the proposed scheme, we can significantly reduce unnecessary flash operations through replacing costly full merges with low-cost switch merges. For this enhanced optimization, we need the cooperation between both the software layers, BML and FTL, which will be explained in detail in the following subsections.

5.1.1 Selective Block Padding

We propose a new BML policy called *Selective Block Padding* (SBP) for co-optimization with BAST. In this policy, a victim buffer block is selected in block-level LRU order, and LRU compensation of BPLRU [11] is adopted to prevent sequential writes from unnecessarily occupying the buffer cache. In our approach, BML can be aware of the state information



(a) Buffer flush without the CO-OP scheme



(b) Buffer flush with the CO-OP scheme

Fig. 4 Buffer flushing examples with and without the CO-OP scheme in BAST

of log blocks in FTL such as the number of written pages and how they were written, sequentially or non-sequentially, by using the CO-OP interfaces. With this information, SBP selectively applies the padding technique. When BML flushes a victim buffer block, it decides whether to pad the victim block, depending on the state of the corresponding log block. If the log block can accommodate the flush without merge operations or will be merged by a switch merge after the flush, BML flushes the victim block without padding to minimize the padding overhead. If the log block must be merged by a full merge through the flush, BML pads the victim block to furnish FTL with a condition for avoiding the full merge.

Algorithm 1 describes the algorithm of the proposed SBP policy that exploits the CO-OP interfaces in Table 3. Blk_{vic} denotes a victim buffer block, and $\mathcal{P}_d(Blk_{vic})$ is the collection of dirty pages in Blk_{vic}. If the number of dirty pages in the victim block is smaller than the number of empty pages in the corresponding log block ($|\mathcal{P}_d(Blk_{vic})| < N_{freePG}$) or if the log block has not been allocated yet ($N_{freePG} = \emptyset$), SBP flushes the victim block without padding. In this case, the dirty pages are simply written to the corresponding log block in FTL, and no merge operation occurs in the log block. In particular, if $\mathcal{P}_d(Blk_{vic})$ fits into the empty pages of the log block and the log block was written in sequence (Is_Seq_Log(LBN of Blk_{vic}) = True), SBP also does not pad the victim block to remove unnecessary padding overhead. In this case, FTL simply converts the log block to a new data block by a switch merge. In the other cases, SBP flushes the victim block (Blk_{vic}) after padding it to produce a complete-block flush since a full merge is anticipated.

SBP also considers an unfavorable scenario in BAST, where one of the pre-allocated log blocks must be reclaimed by a full or partial merge if BML flushes a victim block

Algorithm 1 Selective Block Padding for BAST

1: **procedure** FLUSH_BUFFER(Blk_{vic})
2: // *LBN*: *Logical Block Number*
3: // *NPB*: *Number of Pages per Block*
4: $N_{freePG} \Leftarrow$ **Get_NFreePG_Log**(LBN of Blk_{vic})
5: **if** ($N_{freePG} \neq \emptyset$) **then**
6: **if** ($|\mathcal{P}_d(Blk_{vic})| < N_{freePG}$) **then**
7: Flush Blk_{vic} without padding
8:
9: **else if** ($|\mathcal{P}_d(Blk_{vic})| > N_{freePG}$) **then**
10: Flush Blk_{vic} with padding
11:
12: **else if** (First dirty page offset of Blk_{vic} = NPB $- N_{freePG}$)
13: $\wedge$ (**Is_Seq_Log**(LBN of Blk_{vic}) = True) **then**
14: Flush Blk_{vic} without padding
15: **else**
16: Flush Blk_{vic} with padding
17: **end if**
18: **else if** ($N_{freePG} = \emptyset$) **then**
19: $N_{freeLog} \Leftarrow$ **Get_NFreeLog**()
20: $LBN_{vicLog} \Leftarrow$ **Get_Victim_Log**()
21: $Blk_{vicLog} \Leftarrow$ Find_Buffer(LBN_{vicLog})
22:
23: **if** ($N_{freeLog} = 0 \wedge Blk_{vicLog} \neq \emptyset$) **then**
24: Flush Blk_{vicLog} with padding
25: **end if**
26: flush Blk_{vic} without padding
27: **end if**
28: **end procedure**

(Blk_{vic}) that holds no log block ($N_{freePG} = \emptyset$) and if there is no available free block in FTL ($N_{freeLog} = 0$). In addition, if dirty pages that correspond to this reclaimed log block are cached in the buffer cache and flushed shortly, another pre-allocated log block should be reclaimed again to allocate a new log block. To remove full or partial merges caused by the repeated log block redemption in this scenario, before flushing the victim block (Blk_{vic}), SBP pads and flushes the buffer block (Blk_{vicLog}) that contains dirty pages for the expected victim log block. For this additional optimization, BML should recognize the number of free log blocks and LBN of the expected victim log block, which are also given by the CO-OP interfaces. In the following subsection, we explain how the underlying FTL manages those complete-block flushes generated by SBP in order to avoid costly full merges.

5.1.2 Optimized Switch Merge

The BAST-based FTL is also optimized for the BML behavior by installing a new merge technique called *Optimized Switch Merge* (OSM). On a write request, FTL can determine whether the write request is a complete-block flush by checking the first-page offset and length without an additional interface. If FTL receives a complete-block flush from BML to one of the pre-allocated log blocks, it writes the buffer flush to a free block instead of

the pre-allocated log block, and the fully-written free block becomes the new data block. Thereafter, the corresponding data and the pre-allocated log blocks are simply reclaimed by two erase operations. Consequently, FTL replaces a costly full merge with a lightweight switch merge.

In addition, FTL equipped with the OSM technique obtains a completely empty log block at the end of the buffer flush, which provides an additional benefit that raises the possibility of making switch merges. In cases where the corresponding log blocks have not been allocated yet, original switch merges take place when FTL receives complete-block flushes.

5.1.3 Cost analysis of the CO-OP scheme: SBP and OSM

In this section, we analyze the cost of the CO-OP scheme, compared with those of BPLRU [11], FAB [13], BP-REF [14], BA-GC [15]. From this analysis, we can also directly compare the proposed technique with the existing unconditional and conditional padding techniques.

For the following analysis, we use the notations described in Table 2. N_i is defined as the number of dirty pages of a victim buffer block, and N_r means the number of remaining empty pages of the corresponding log block, simply called the log block, in FTL. When flushing a victim block that contains N_i dirty pages to the log block that holds N_r remaining empty pages, we compute the costs of the five different schemes. C_{FAB}, C_{BPLRU}, and $C_{BP\text{-}REF}$ mean the costs of the non-padding, unconditional block padding, and conditional block padding policies, respectively. $C_{BA\text{-}GC}$ is the result when using the BA-BM technique explained in Sect. 3, and $C_{CO\text{-}OP}$ is caused by both the proposed techniques, SBP and OSM. Note that in the cases of $N_i < N_r$ or switch merges, we can simply say: $C_{CO\text{-}OP} = C_{FAB} = C_{BA\text{-}GC} \le C_{BPLRU}, C_{BP\text{-}REF}$. Considering random write patterns, we assume that the log block is reclaimed by a full merge for $N_i \ge N_r$ and $N_r > 0$. The cost of each policy can be formulated as follows:

$$
\begin{aligned}
C_{FAB} &= N_r \cdot T_w + GC_{FM} + (N_i - N_r) \cdot T_w + C_{pot(N_i - N_r)} \\
&= N_i \cdot T_w + GC_{FM} + C_{pot(N_i - N_r)} \\
\big(C_{pot(N_i - N_r)} &= GC_{FM} \cdot (N_i - N_r)/N\big) \\
C_{BPLRU} &= C_{prev(N-N_r)} + C_{pad(N-N_i)} + N \cdot T_w + GC_{SM} \\
\big(C_{prev(N-N_r)} &= N_r \cdot (T_r + T_w) + GC_{SM}\big) \\
\big(C_{pad(N-N_i)} &= (N - N_i) \cdot T_r\big) \\
C_{BP\text{-}REF} &= C_{pad(N-N_i)} + N_r \cdot T_w + GC_{FM} + (N - N_r) \cdot T_w + C_{pot(N-N_r)} \\
\big(C_{pad(N-N_i)} &= (N - N_i) \cdot T_r\big) \\
\big(C_{pot(N-N_r)} &= GC_{FM} \cdot (N - N_r)/N\big) \\
C_{BA\text{-}GC} &= N_r \cdot T_w + GC_{BA\text{-}BM} \\
\big(GC_{BA\text{-}BM} &= GC_{FM} - (N_i - N_r) \cdot T_r\big) \\
C_{CO\text{-}OP} &= C_{pad(N-N_i)} + N \cdot T_w + GC_{OSM} \\
\big(C_{pad(N-N_i)} &= (N - N_i) \cdot T_r\big) \\
(GC_{OSM} &= 2 \cdot T_e)
\end{aligned}
$$

In FAB as the non-padding policy, the N_r remaining empty pages of the log block are fully written by a part of dirty pages of the victim block. Thereafter, the log block is reclaimed by a full merge, and the remaining $N_i - N_r$ dirty pages of the victim block are written to the newly-allocated log block. As a result, after the victim block is flushed, the new log block is written as many as $N_i - N_r$ pages, antedating the following merge operation. Accordingly, we additionally compute the potential overhead as $C_{pot(N_i - N_r)}$, which is derived from the $N_i - N_r$ page writes to the new log block.

In BPLRU using the unconditional padding technique, N_r cannot be more than zero since BPLRU triggers only switch merges. For fair comparison and making N_r zero, we include the padding and switch merge overheads ($C_{prev(N-N_r)}$) for the previously flushed $N - N_r$ pages (N_{i-1}), assuming that N_f was 1. Note that the cost of BPLRU can be considerably larger with a low dirty rate having large N_f as explained in Sect. 4. To flush N_i dirty pages, BPLRU pads the omitted $N - N_i$ pages ($C_{pad(N-N_i)}$) and flushes a complete block ($N \cdot T_w$). Then, the log block is merged by a switch merge (GC_{SM}). Since BPLRU obtains completely empty log block after the flush of N_i dirty pages, there is no potential overhead.

In BP-REF, we should carefully consider the conditional block padding technique. In this analysis, we assume that BP-REF did not pad the previous flush of $N - N_r$ pages (N_{i-1}) but pads the current flush of N_i pages. For the current flush, BP-REF fills up the omitted pages of the victim block ($C_{pad(N-N_i)}$), and flushes some of dirty pages to the remaining N_r empty log pages. After the log block is reclaimed by a full merge (GC_{FM}), the remaining dirty pages are flushed to the newly-allocated log block, generating the potential overheads ($C_{pot(N-N_r)}$).

If BP-REF pads both the flushes, the cost of BP-REF becomes the same as that of BPLRU. Conversely, if both the flushes are not padded, the cost is the same as that of FAB. If BP-REF pads only the previous flush, the cost just after the current flush will change according to N_r and N_i. From the next flush, however, the cost will follow this analysis because the log block is already occupied by the N_i written pages.

In BA-GC, the N_r remaining empty pages of the log block are fully written by a part of dirty pages of the victim block like the FAB policy. Then, the log block is merged by BA-BM whose cost is $GC_{BA\text{-}BM}$. During the BA-BM operation, since the remaining $N_i - N_r$ dirty pages of the victim block can be read from the buffer cache for valid page migration, $N_i - N_r$ page reads of NAND flash memory are saved, compared with GC_{FM}. After this merge operation, the remaining $N_i - N_r$ dirty pages of the victim block are already changed to clean pages. Then, these remaining clean pages of the victim block are simply evicted from the buffer cache without flushing, and thus $C_{BA\text{-}GC}$ can be lower than C_{FAB}.

Finally, in CO-OP, SBP pads the victim block by reading the $N - N_i$ omitted pages from FTL due to $N_r \leq N_i$, and then reclaims the log block and its data block by OSM, whose cost is GC_{OSM}, after simply writing the complete block to a free block. The following inequality describes the comparison of the costs generated by five different schemes.

$$
\begin{aligned}
&C_{CO\text{-}OP} < C_{BPLRU} (= C_{BA\text{-}GC}) \leq C_{FAB} \leq C_{BP\text{-}REF} \\
&\qquad = 0 < N_r \cdot (T_r + T_w) \leq N_i \cdot (T_r + T_w) + C_{pot(N_i - N_r)} \leq N \cdot (T_r + T_w) + C_{pot(N - N_r)}
\end{aligned}
$$

From this analysis, when flushing a victim block of the buffer cache for $N_r \leq N_i$ and $N_r > 0$, we demonstrate that $C_{CO\text{-}OP}$ always shows the lowest cost, compared with the other schemes.

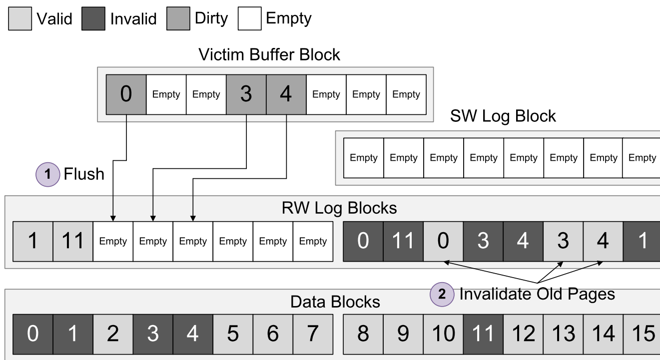
5.2 Case study with Fully-Associative Sector Translation (FAST)

The CO-OP scheme including the SBP and OSM techniques can be also employed in FAST to enhance the performance. The proposed CO-OP scheme presents FAST a more intelligent

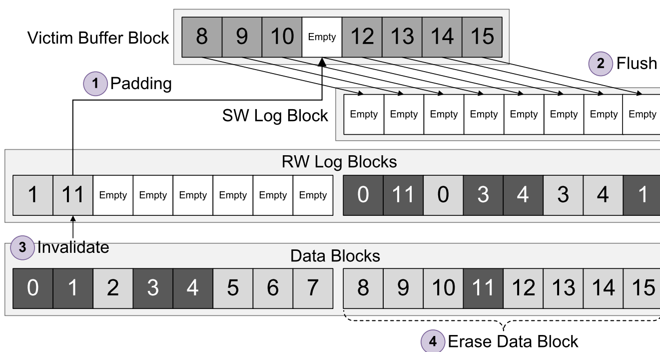
criterion for redirecting buffer flushes to RW log blocks or the SW log block. This co-optimization is an advanced approach to reduce partial merges in the SW log block as well as full merges in RW log blocks.

To reduce full merges in FAST, random writes that exhibit high temporal locality should be stored in RW log blocks. Such random writes are likely to be invalidated in RW log blocks by following updates. If all valid pages that belong to the same associated data block in an RW log block are invalidated, FAST can avoid a full merge for the associated data block when reclaiming the RW log block. We call this a Full Merge (FM) hit. From preliminary workload analysis with the traces in Table 5, we observed that buffer flushes that contain a small number of dirty pages can make more FM hits in RW log blocks. In the existing FAST algorithm, however, RW log blocks are occupied by various-sized write requests, lowering the FM hit ratio. Furthermore, FAST is vulnerable to multiple sequential write streams and the mixture of random and sequential write patterns, since FAST maintains only one SW log block. For example, if there are many small random writes whose first-page offset is zero, the SW log block is repeatedly reclaimed by numerous partial merges.

Considering those problems in FAST, the CO-OP scheme distinguishes random and sequential writes by means of the number of dirty pages in a victim buffer block to remove expensive full and partial merges. Figure 5 shows the examples of flushing processes based on the CO-OP scheme. In this example, the SBP technique selectively pads the victim buffer



(a) How to handle random writes in the CO-OP scheme



(b) How to handle sequential writes in the CO-OP scheme

Fig. 5 Buffer flushing examples with the CO-OP scheme in FAST

block according to the number of dirty pages. As shown in Fig. 5(a), the CO-OP scheme flushes the victim block without padding if the number of dirty pages is within the Random Write (RW) threshold. The FTL equipped with the OSM technique writes the buffer flush to RW log blocks from regarding the flush as random writes. This is helpful for reducing partial merges and unnecessary padding overheads and also for increasing the FM hit ratio. Note that, in the Fig. 5(a), the 0 dirty page in the victim block should be written into the SW log block without OSM. As shown in Fig. 5(b), if the number of dirty pages is larger than the RW threshold, SBP pads the omitted pages of the victim block, and FTL flushes the complete-block to the SW log block. After this, the related log pages and the data block are invalidated and erased, respectively. In this way, the underlying FTL can redirect a write request into RW or SW log blocks, depending on whether the request is a complete-block flush, not on the first-page offset of the request.

5.2.1 Selective Block Padding

Algorithm 2 shows the algorithm of *Selective Block Padding* (SBP) for FAST. When flushing a victim block (Blk_{vic}) from the buffer cache, BML checks whether the number of dirty pages in the victim block is larger than the RW threshold ($Thres_{RW}$), which is obtained by the CO-OP interface (Get_Thres_RW()). If the number of dirty pages is within the RW threshold ($Thres_{RW}$), BML flushes the victim block without padding. Otherwise, BML flushes the victim block after padding it to provide a complete-block flush for FTL. In the next subsection, we explain how the underlying FTL exploits these write patterns generated by the SBP technique as a co-optimization.

Algorithm 2 Selective Block Padding for FAST

```
1: procedure FLUSH_BUFFER(Blk_vic)
2:     Thres_RW ⇐ Get_Thres_RW()
3:     if (# of dirty pages in Blk_vic ≤ Thres_RW) then
4:         Flush Blk_vic without padding
5:     else
6:         Flush Blk_vic with padding
7:     end if
8: end procedure
```

5.2.2 Optimized Switch Merge

For the cooperative optimization with BML, *Optimized Switch Merge* (OSM) should be also adopted in the existing FAST algorithm. The FTL equipped with the OSM technique writes a complete-block flush, which denotes sequential writes, to the SW log block, which is simply reclaimed by a switch merge. In the other cases, buffer flushes are redirected to RW log blocks, considered as random writes. These filtered random writes can be more effective for increasing the FM hit ratio in the RW log blocks. Moreover, the complete-block flushes always trigger switch merges in the SW log block, thus eliminating costly partial merges.

For this co-optimization between the SBP and OSM techniques, FTL should offer the RW threshold, which is demanded by the CO-OP interface (Get_ Thres_RW( )). In the proposed CO-OP scheme, the initial RW threshold is set as 70 when the number of pages per block is 128, and the RW threshold is dynamically adjusted considering the FM hit ratio and workload characteristics.

6 Performance evaluation

We evaluated the CO-OP scheme with two widely-used FTL algorithms: BAST and FAST, compared with the existing BML and FTL schemes summarized in Table 4. Although the BA-GC scheme was expounded only for FAST in the paper [15], we modified and adopted the BA-GC algorithm properly for BAST, based on the comment from the first author. In the modified algorithm, when a victim log block is selected by BA-VBS, a log block that reserves enough free pages for accommodating the dirty buffer pages to be flushed is excepted from candidates to avoid unnecessary merge operations. To clearly compare the proposed scheme with the non-padding policy, we additionally define BLRU, which is identical with BPLRU [11] without the block padding technique.

6.1 Simulation configurations

The following simulation is configured for MLC NAND flash memory [2] that provides a larger capacity and a cheaper price than SLC NAND flash memory [1], and that is more suitable for consumer-based flash storage devices. The specifications of both types of NAND flash memory are described in Table 1. According to the previous studies [21, 22], the portion of extra blocks is configured as 3% of the overall NAND flash memory whose size is 64 GB.

The DRAM size ranges from 4 MB to 32 MB. A part of the DRAM is used for the FTL mapping cache, and the remaining part is used for the buffer cache. For example, BPLRU can utilize almost all of the DRAM space for the buffer cache since it requires few log blocks. As previously mentioned, the buffer cache is used only for write caching, not for read caching [11, 15].

We implemented a trace-driven simulator that operates with the various block-level traces, which are summarized in Table 5. We classify the traces into three groups. First, Pic and MP3 traces model the workloads of mobile embedded devices such as digital cameras, MP3 players, and Portable Media Players (PMPs). To generate these workloads, we

Table 4 Summary of the existing BML and FTL schemes

	Target	Buffer management policy	Victim unit	BML flushing policy	FTL merge algorithm
FAB [13]	BML	Largest block-based victim selection + block-level LRU	Block	Non-padding	Unmodified
BPLRU [11]	BML	Block-level LRU + LRU compensation	Block	Unconditional block padding	Unmodified
BP-REF [14]	BML	Page-level LRU + victim window	Block/page	Conditional block padding	Unmodified
BA-GC [15]	FTL	3-region LRU	Block	Non-padding	BA-VBS BA-BM
CO-OP	BML + FTL	Block-level LRU + LRU compensation	Block	Selective block padding	Optimized switch merge

Table 5 Summary of the block-level traces that model various workloads

Name	Description	Req. ratio [read/write] (%)	Req. size [read/write] (KB)	Working set [read/write] (GB)
Pic	Copying and deleting picture files repeatedly (avg. file size = 1.9 MB)	–/100	–/55.8	–/1.69
MP3	Copying and deleting MP3 files repeatedly (avg. file size = 4.4 MB)	–/100	–/63.1	–/1.89
Multi	Multimedia-based works by using web camera applications, video editors, audio editors, photoshop, 3D Maya, etc.	43.6/56.4	10.1/19.6	15.4/6.8
Web	Web-based works such as web browsing, audio playing, file downloading, and running email clients and messengers	24.5/75.5	7.8/9.2	6.5/5.2
TPC-C	Running the open-source TPCC-UVa benchmark that creates and tests a new database [31]	1.4/98.6	20.6/9.5	0.1/1.2
HM_Server	Hardware monitoring function in data center servers [32]	35.5/64.5	7.4/8.3	1.9/1.6

repeatedly deleted and created 2 GB worth of picture or MP3 files 10 times for each trace after creating 6 GB worth of the multimedia files. Second, we employed two realistic workloads from common desktop systems such as the Multi and Web traces. Finally, we used server workloads such as the TPC-C and HM_Server traces to evaluate the performance in On-Line Transaction Processing (OLTP) environments and in data center servers. The mobile embedded workloads mainly consist of sequential writes, while the other workloads contain many random writes with some temporal locality. These three-grouped traces were helpful to evaluate the performance of the proposed techniques under the various workloads of mobile embedded, desktop, and server systems.

6.2 Throughput

We measured the throughput (KB/s), which means the rate of the total amount of read and written data to the overall operation time, by using all the traces with different DRAM sizes from 4 MB to 32 MB. As shown in Figs. 6 and 7, CO-OP outperforms the competitors in most cases. For instance, when the DRAM size is 16 MB in the Multi trace with FAST, the throughput is improved by 55% over that of BLRU, the next closest competing scheme.

As shown in the mobile embedded traces of Fig. 6(a), (b) and Fig. 7(a), (b), most schemes show similar performance with more than 16 MB DRAM, since these workloads consist of many sequential writes and a few random writes. However, we observed that FAB and BP-REF exhibit poor performance with less than 16 MB DRAM. In FAB, a buffer block that holds the largest number of valid pages is selected as a victim block in preference to comparing temporal locality. Accordingly, in the buffer cache, *sequential blocks* that contain sequential writes are likely to face a higher probability of being evicted than *random*

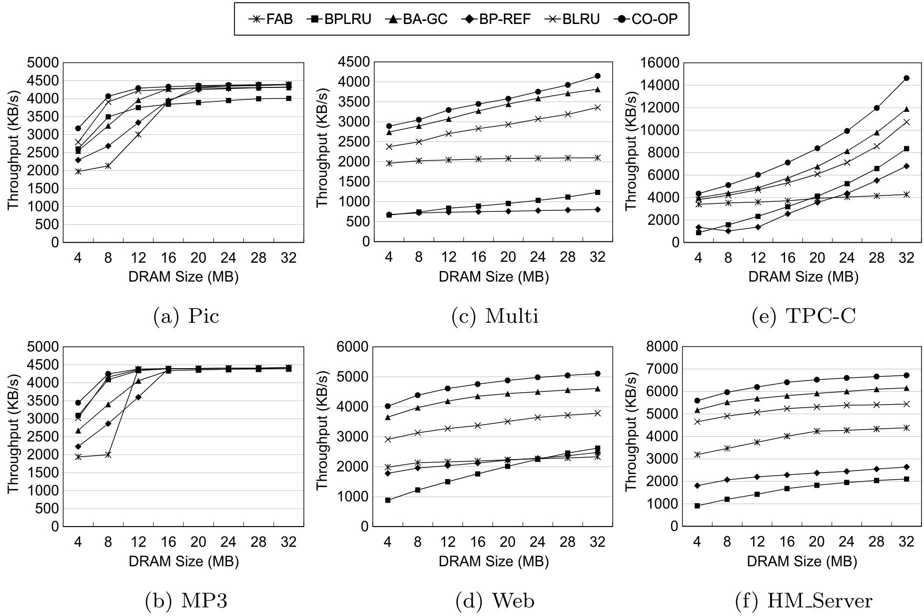


Fig. 6 Throughput comparison under various workloads with BAST

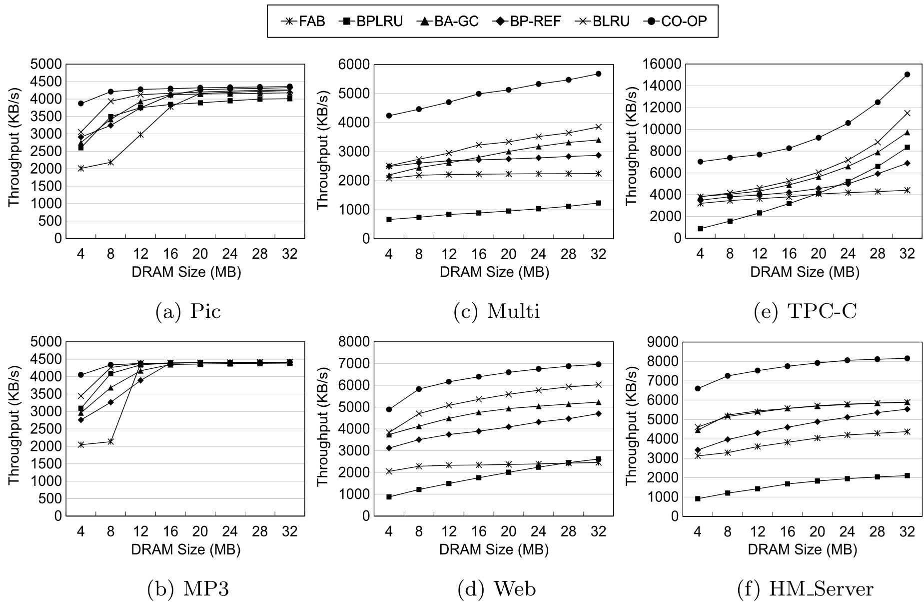


Fig. 7 Throughput comparison under various workloads with FAST

blocks that contain random writes. If the buffer space is insufficient, these random blocks thoroughly occupy the buffer cache with a lower probability of being evicted. In this circumstance, sequential blocks tend to be flushed early due to their larger dirty pages before they

are fully filled with following sequential writes and become complete blocks. Furthermore, even sequential blocks being currently written can be victim blocks without considering their temporal locality. These early-evicted sequential blocks become the main cause of full merges in FTL, which we call an *early eviction problem*. Since BP-REF employs the FAB-like victim selection policy within the victim window, it suffers the same problem, which also can involve unnecessary padding overheads. However, if the buffer cache accommodates most random writes and leaves enough space for gathering sequential writes, FAB and BP-REF can avoid this problem as shown in the results with more than 16 MB DRAM of Figs. 6(a), (b) and 7(a), (b).

Unlike FAB and BP-REF, there is no such a problem in BPLRU, BLRU, and CO-OP since they are based on the block-level LRU replacement policy. In this policy, sequential writes can be freely assembled into complete blocks in the buffer cache. In addition, by means of LRU compensation, these complete blocks can be flushed to FTL preferentially, supporting lightweight switch merges.

In the random workloads of the desktop and server groups, BA-GC exhibits better performance than BLRU in the results of Fig. 6(c)–(f) with BAST in accordance with the preceding analysis of Sect. 5.1.3. In Fig. 7(c)–(e) with FAST, however, the performance of BA-GC sinks more or less below that of BLRU, the non-padding policy, due to the lower buffer hit ratio of the 3-region LRU policy and the lower FM hit ratio in RW log blocks. In these workloads, since the working set size of write patterns is larger than the amount of log blocks unlike the other workloads, there is an additional behavior where one of pre-allocated log blocks must be reclaimed whenever there are no more free blocks. For this behavior, in the BA-VBS technique of BA-GC, a log block that holds many dirty and cold pages in the buffer cache is selected as a victim in FTL regardless of the temporal locality of the log block. Practically, in FAST, an RW log block that keeps many dirty pages in the buffer cache can be invalidated by flushing the dirty pages and thus can reduce full merges. In this way, the log pages that contain hot data can be repeatedly invalidated within RW log blocks without full merges because these random workloads exhibit some temporal locality for write requests. However, if the RW log block is selected as a victim by BA-VBS, it should be reclaimed by full merge-based BA-BM. In other words, the BA-VBS technique of BA-GC can select an RW log block to be naturally invalidated as a victim in FTL instead of the LRU RW log block, lowering the FM hit ratio.

In all the results under the random workloads shown in Figs. 6(c)–(f) and 7(c)–(f), CO-OP always achieves better performance than the other schemes including BLRU, while BPLRU and BP-REF deliver similar or lower performance than BLRU. These results confirm that the unconditional or conditional padding technique involves large padding overheads under random write patterns, but CO-OP ensures stable performance enhancement even under random write patterns with the different DRAM sizes.

6.3 FTL merge overhead

Figures 8 and 9 present the breakdowns of the merge count and cost when the DRAM size is 16 MB with BAST and FAST, respectively. The overall merge count and cost for each scheme are normalized to the largest value among the six schemes. In this breakdown, there are three types of fundamental merge operations such as switch, partial, and full merges, and three types of particular merge operations such as *Padding Switch Merge* (PSM), BA-BM, and OSM for the BPLRU, BA-GC, and CO-OP schemes, respectively.

For BPLRU and BP-REF, we classify the switch merges into two groups, original switch merges and PSMs artificially made by the unconditional or conditional block padding technique. In CO-OP, the count of OSM increases when a complete block is flushed from the

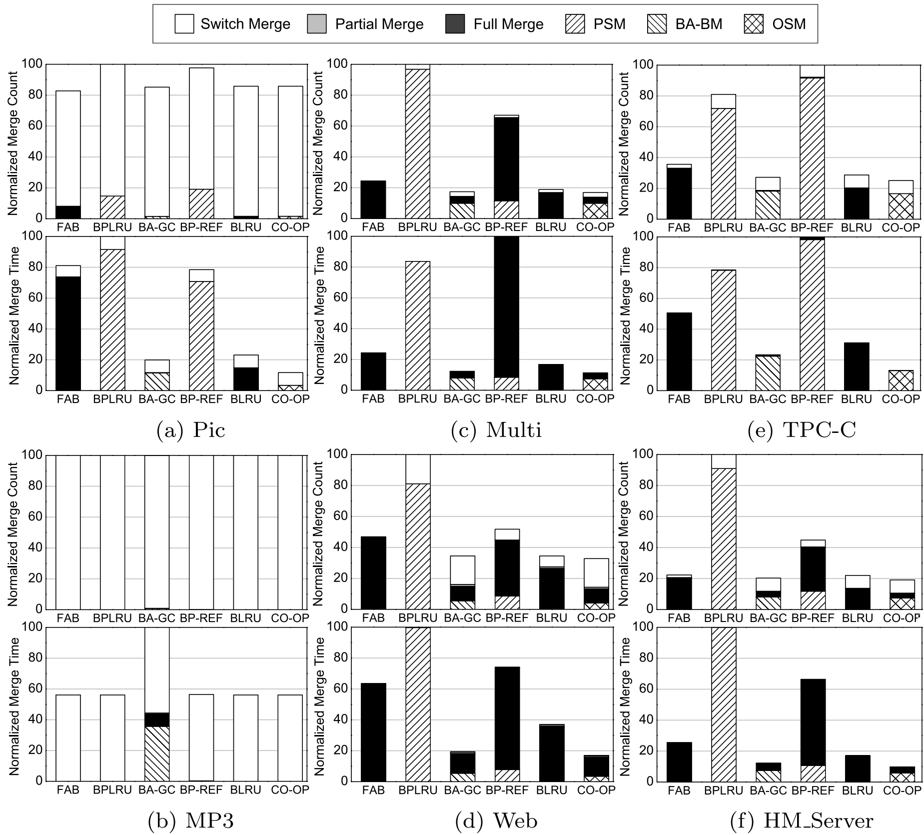


Fig. 8 Breakdown of FTL merges: normalized merge count and merge cost with BAST

buffer cache after being padded to avoid full or partial merges. In BPLRU, BP-REF, or CO-OP, the padding overhead that involves flash reads and writes is included in the cost of PSM, full merges, or OSM. In BA-GC, BA-BM is counted when FTL reads valid pages from the buffer cache or changes the state of dirty buffer pages into clean for each merge operation. The write cost, as many as the number of these dirty buffer pages, is excluded in the cost of BA-BM for fair comparison.

FAB presents many full and partial merges in the breakdowns even under the sequential workload shown in Figs. 8(a) and 9(a) due to the early eviction problem where sequential blocks are shoved by random blocks in the buffer cache as explained in the previous subsection. For the random workloads in Figs. 8(c)–(f) and 9(c)–(f), full merges also occupy a large portion of the breakdowns since there are few complete-block flushes. That is why a multitude of full or partial merges augment the total merge cost. Note that few full merges occur with FAST under some workloads that present a small working set size in Fig. 9(a), (b), (e), because there is no full merge operation until all RW log blocks are consumed.

BPLRU generates mainly PSMs and switch merges without full and partial merges, but for most of the results it takes the largest number of merge operations, which is identical with the number of flushes. In the sequential workload, the cost of PSM is relatively low due to small padding overheads, while, in the random workloads, numerous PSMs with

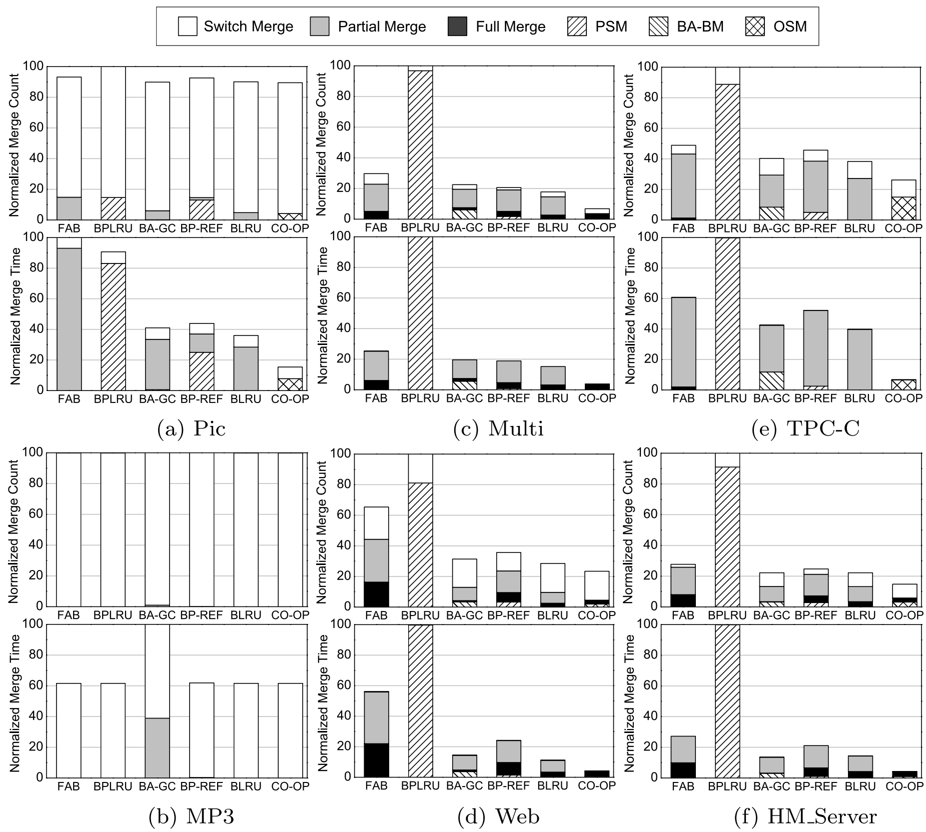
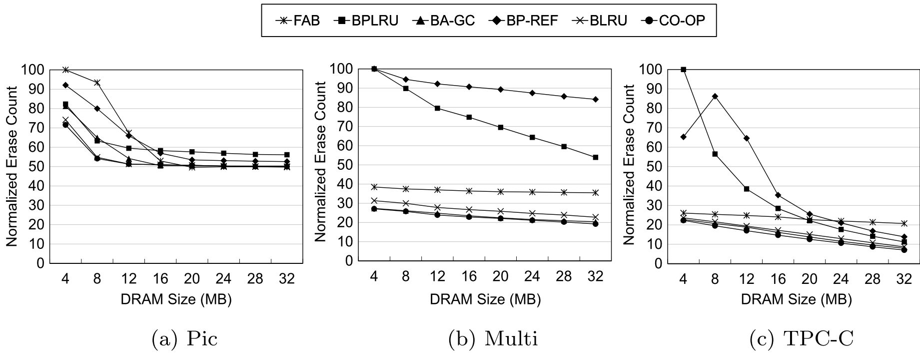


Fig. 9 Breakdown of FTL merges: normalized merge count and merge cost with FAST

large padding overheads significantly raise the overall merge cost of BPLRU beyond those of the competitors.

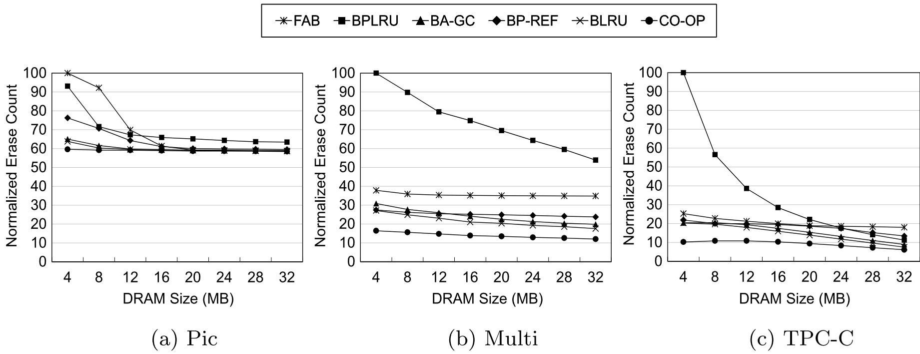
BP-REF shows a smaller merge cost than BPLRU with FAST, but there is a critical problem under random write patterns with BAST. If BP-REF pads and flushes a victim buffer block to the log block that already contains written pages, a full merge occurs instead of a switch merge, also encountering the padding overhead. Therefore, the merge cost of BP-REF is considerably larger than that of BLRU as the non-padding policy, and we call this a *conditional padding problem*, which was previously analyzed in Sect. 5.1.3. Even with FAST under most of the workloads, the performance of BP-REF falls behind that of BLRU due to large padding overheads caused by the unawareness of the underlying FTL state.

Compared with BLRU, CO-OP replaces many full merges in BAST and all of the partial merges in FAST with lightweight OSMs, extending opportunities for reducing the overall merge count and cost. In particular, CO-OP entirely removes full merges with sufficient log blocks in BAST as shown in Fig. 8(e). As shown in Fig. 8(d), with BAST, CO-OP also can increase the number of original switch merges because it obtains completely empty blocks after victim log blocks are reclaimed by OSM. Moreover, in FAST, CO-OP eliminates all of the frequent partial merges by exploiting the SBP and OSM techniques, which is a key factor of the significant performance enhancement.



(a) Pic (b) Multi (c) TPC-C

Fig. 10 Comparison of the number of erase operations with BAST



(a) Pic (b) Multi (c) TPC-C

Fig. 11 Comparison of the number of erase operations with FAST

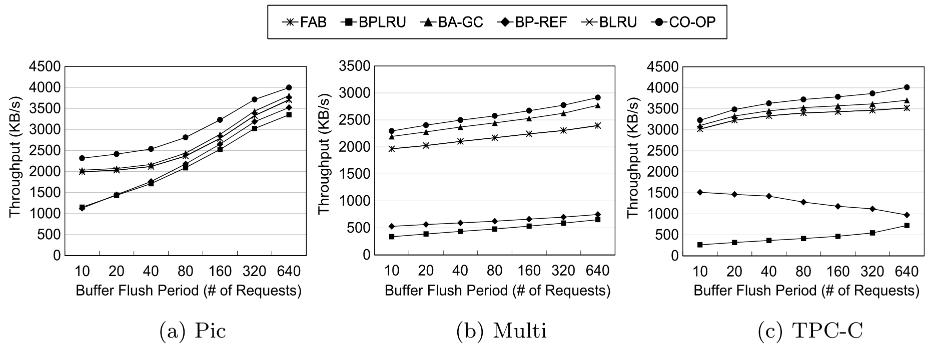
Although the merge costs between BA-GC and the other schemes are quite different in the MP3 trace shown in Figs. 8(b) and 9(b), there is no great difference with the overall performance among all the competitors, because the portion of the merge cost is less than just 3% of the total operation time in that trace unlike the other traces.

6.4 Erase count of NAND flash memory

The erase count of NAND flash memory is an important factor that directly affects the lifetime as well as the performance of flash storage devices. Figures 10 and 11 show the number of erase operations in the three traces from each workload group according to the DRAM size; each result is normalized to the largest count.

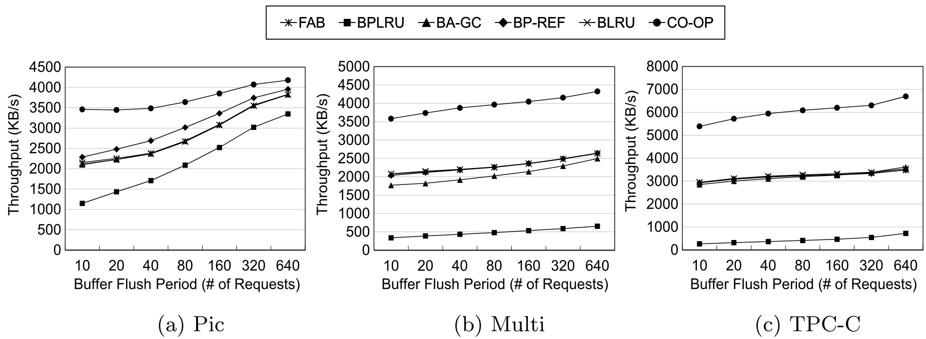
In most of the results, there are many erase operations when employing BPLRU or BP-REF. Although BPLRU results in only switch merges each of which consists of one block erase, the merge count is significantly large. Exceptionally, the erase count is rather increased from 4 MB to 8 MB in Fig. 10(c) with BP-REF. This is because the conditional padding technique based on the predefined threshold unnecessarily causes more of the conditional padding problems when using the larger buffer cache.

Under the sequential workloads shown in Figs. 10(a) and 11(a), FAB generates a large number of erase operations with the small buffer size due to the early eviction problem, since it incurs many full or partial merges. Under the random workloads, the write hit ratio



(a) Pic (b) Multi (c) TPC-C

Fig. 12 Effects of the periodic flush cache command with BAST



(a) Pic (b) Multi (c) TPC-C

Fig. 13 Effects of the periodic flush cache command with FAST

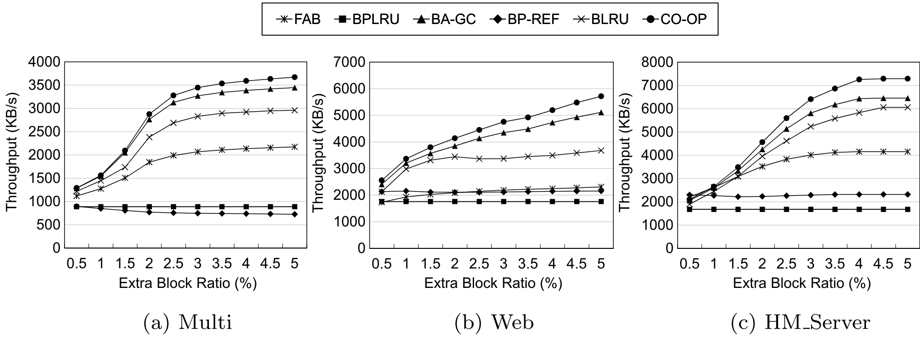
of the buffer cache in FAB and BP-REF is lower than the other schemes because their victim selection policy is based on the number of dirty pages for each buffer block prior to temporal locality. These write requests missed from the buffer cache also increase the erase count.

Considering CO-OP provides the smallest erase count under the various workloads and DRAM sizes with BAST and FAST, we anticipate that the proposed scheme helps flash storage devices extend the lifetime as well as enhance the performance comparatively.

6.5 Effect of flush cache commands

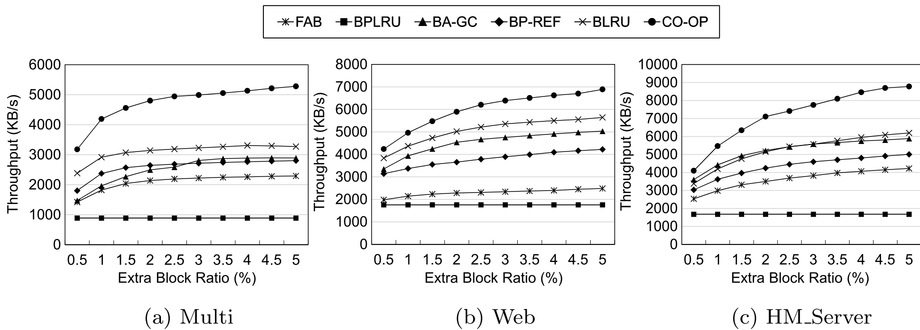
To prevent the loss of write-buffered data in the non-volatile buffer cache under unexpected power failures, a flush cache command to the flash storage device can be periodically issued by the host or by itself. Whenever the flush cache command is received, the buffer cache is completely flushed. If the frequency of the flush cache command is very high, write requests are likely to be flushed before being gathered in the buffer cache, so that there are almost no write buffering and caching benefits. To analyze the effects of the periodic flush commands, we measured the throughput (KB/s) under different flush command intervals from 10 to 640 requests with three traces from each workload group with 16 MB DRAM.

As shown in Figs. 12(a)–(c) and 13(a)–(c), the performance of CO-OP outperforms that of the other schemes in all of the results. In the Multi and TPC-C traces with BAST shown in Fig. 12(b), (c), the performance gap between CO-OP and FAB or BLRU narrows more



(a) Multi (b) Web (c) HM_Server

Fig. 14 Performance according to the number of extra blocks with BAST



(a) Multi (b) Web (c) HM_Server

Fig. 15 Performance according to the number of log blocks with FAST

than the performance gap without the flush cache command. This is because these workloads consist of small random writes that reduce N_i of buffer flushes, which was explained in Sect. 5.1.3. In the other results, however, the performance gap is more widely extended because frequent full and partial merges create many opportunities to improve performance for CO-OP. From these results, we expect that the proposed scheme can reliably enhance the performance even under frequent flush cache commands.

6.6 Effect of the log block ratio

By adjusting the log block (extra block) ratio up to 5% of the overall flash capacity, we measured the throughput (KB/s) when the DRAM size is 16 MB under the three realistic workloads, which exhibit a large working set size. In the other workloads with a small working set size, the performance difference between the competitors shows only slight changes according to the log block ratio.

As shown in Figs. 14 and 15, CO-OP outperforms the other schemes not only with a small number of log blocks but also with a large number of log blocks. Moreover, the performance gap with the competitors is more widened as the log block ratio increases. In BAST, CO-OP can create more opportunities for lightweight OSMs with sufficient log blocks because there is no log block thrashing where pre-allocated log blocks are repeatedly reclaimed. Also, in FAST, CO-OP can increase the FM hit ratio with enough log blocks by cooperatively dis-

tinguishing random and sequential writes. However, the other schemes do not show notable performance enhancement even with sufficient log blocks especially in FAST.

In particular, BPLRU always obtains the same performance under different log block ratios because its policy is not affected by the number of log blocks. Since BPLRU produces only switch merges, most of the log blocks remain as free blocks, and the unused mapping space in DRAM is utilized for write caching. If the number of log blocks is extremely small, BPLRU may be relatively more cost-beneficial than the other schemes. However, this means a strict limit on elevating the performance.

In Fig. 14(a) with BAST, the performance of BP-REF is rather reduced as the log block ratio increases, because of the conditional padding problem previously mentioned in Sect. 6.3. Paradoxically, if there are a small number of log blocks, the problem is alleviated because padded victim blocks are likely to be flushed to data blocks that do not hold pre-allocated log blocks. From this evaluation, we can confirm that the proposed scheme achieves the best performance even in a wide range of the log block ratio.

7 Conclusion

Recently, we have witnessed the rapid growth of NAND flash-based storage devices as notable secondary storage devices. To enhance their performance, we focused on the cooperative optimization (CO-OP) of two major software layers: Buffer Management Layer (BML) and Flash Translation Layer (FTL). For improved cooperation between those two layers, we proposed two main techniques: an FTL-aware BML policy called *Selective Block Padding* (SBP) and a BML-assisted FTL algorithm called *Optimized Switch Merge* (OSM).

Unlike the existing unconditional or conditional block padding techniques, the SBP technique selectively pads a victim buffer block only when it can derive favorable behavior from FTL by cooperating with FTL in order to minimize extra operations in NAND flash memory. In addition, the OSM technique transforms complete-block flushes supported by SBP into low-cost switch merges instead of costly full or partial merges in FTL. After finishing OSM, we procure completely empty log blocks, which provide additional benefits that further delay upcoming merges and raise the probability of lightweight switch merges in FTL.

To evaluate the performance of the proposed ideas, we set up an analytical model that computes the cost of flushing the buffer cache according to competing approaches, and we also implemented a trace-driven simulator executed by various block-level workloads from mobile embedded, desktop, and server systems. The analysis and simulation results demonstrate that the CO-OP scheme outperforms previous studies even with different DRAM sizes, log block ratios, and periodic flush commands under varying workloads. In this paper, we applied the CO-OP scheme to two widely-used FTL algorithms: BAST and FAST. We expect that the CO-OP scheme can also be further extended to other FTL algorithms like page-level mapping FTLs.

Acknowledgement This work was supported by the IT R&D Program of MKE/KEIT [2010-KI002090, Development of Technology Base for Trustworthy Computing].

References

1. 1 G × 8 Bit/2 G × 8 Bit/4 G × 8 Bit NAND flash memory (K9WAG08U1M) data sheets, Samsung Electronics, Nov 2005
2. 2 G × 8 Bit NAND flash memory (K9GAG08UXM) data sheets, Samsung Electronics, Dec 2006

3. Baek S, Choi J, Ahn S, Lee D, Noh SH (2009) Design and implementation of a uniformity-improving page allocation scheme for flash-based storage systems. Des Autom Embed Syst 13(1–2):5–25
4. Lim S-H, Park KH (2006) An efficient NAND flash file system for flash memory storage. IEEE Trans Comput 55(7):906–912
5. Lee C, Baek SH, Park KH (2008) A hybrid flash file system based on nor and nand flash memories for embedded devices. IEEE Trans Comput 57(7):1002–1008
6. Caulfield AM, Grupp LM, Swanson S (2009) Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In: Proceedings of the 14th international conference on architectural support for programming languages and operating systems (ASPLOS'09), Washington, DC, USA, Mar 2009, pp 217–228
7. Chen F, Koufaty DA, Zhang X (2009) Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: ACM SIGMETRICS/performance, Seattle, WA, USA, Jun 2009, pp 181–192
8. Prabhakaran V, Rodeheffer TL, Zhou L (2008) Transactional flash. In: Proceedings of the 8th USENIX symposium on operating systems design and implementation (OSDI'08), San Diego, CA, USA, Dec 2008, pp 147–160
9. Kang J-U, Kim J-S, Park C, Park H, Lee J (2007) A multi-channel architecture for high-performance NAND flash-based storage system. J Syst Archit 53(9):644–658
10. Agrawal N, Prabhakaran V, Wobber T, Davis JD, Manasse M, Panigrahy R (2008) Design tradeoffs for SSD performance. In: Proceedings of USENIX annual technical conference (USENIX'08), Boston, MA, USA, Jun 2008, pp 57–70
11. Kim H, Ahn S (2008) A buffer management scheme for improving random writes in flash storage. In: Proceedings of the 6th USENIX conference on file and storage technologies (FAST'08), San Jose, CA, USA, Feb 2008, pp 239–252
12. Park S-Y, Jung D, Kang J-U, Kim J-S, Lee J (2006) CFLRU: a replacement algorithm for flash memory. In: Proceedings of the international conference on compilers, architecture and synthesis for embedded systems (CASES'06) held in conjunction with ESWEEK'06, Seoul, Republic of Korea, Oct 2006, pp 234–241
13. Jo H, Kang J-U, Park S-Y, Kim J-S, Lee J (2006) FAB: flash-aware buffer management policy for portable media players. IEEE Trans Consum Electron 52(2):485–493
14. Seo D, Shin D (2008) Recently-evicted-first buffer replacement policy for flash storage devices. IEEE Trans Consum Electron 54(3):1228–1235
15. Lee S, Shin D, Kim J (2008) Buffer-aware garbage collection for NAND flash memory-based storage systems. In: Proceedings of the international workshop on software support for portable storage (IWSSPS'08) held in conjunction with ESWEEK'08, Atlanta, GA, USA, Oct 2008, pp 27–32
16. Ding X, Jiang S, Chen F (2007) A buffer cache management scheme exploiting both temporal and spatial localities. ACM Trans Storage 3(2):5
17. Chiang M-L, Lee PCH, Chang R-C (1999) Using data clustering to improve cleaning performance for flash memory. Softw Pract Exp 29(3):267–290
18. Gupta A, Kim Y, Urgaonkar B (2009) DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In: Proceeding of the 14th international conference on architectural support for programming languages and operating systems (ASPLOS), Washington, DC, USA, Mar 2009, pp 229–240
19. Kim J, Kim JM, Noh SH, Min SL, Cho Y (2002) A space-efficient flash translation layer for compactflash systems. IEEE Trans Consum Electron 48(2):366–375
20. Lee S-W, Park D-J, Chung T-S, Lee D-H, Park S, Song H-J (2007) A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans Embed Comput Syst 6(3):18
21. Kang J-U, Jo H, Kim J-S, Lee J (2006) A superblock-based flash translation layer for NAND flash memory. In: Proceedings of the 6th ACM international conference on embedded software (EMSOFT'06), Seoul, Republic of Korea, Oct 2006, pp 161–170
22. Lee Y-G, Jung D, Kang D, Kim J-S (2008) μ-FTL: a memory-efficient flash translation layer supporting multiple mapping granularities. In: Proceedings of the 8th ACM international conference on embedded software (EMSOFT'08), Atlanta, GA, USA, Oct 2008, pp 21–30
23. Park C, Cheon W, Kang J, Roh K, Cho W (2008) A reconfigurable FTL (Flash Translation Layer) architecture for NAND flash-based applications. ACM Trans Embed Comput Syst 7(4):38
24. Choi HJ, Lim S-H, Park KH (2009) JFTL: a flash translation layer based on a journal remapping for flash memory. ACM Trans Storage 4(4):14
25. Lee J, Kim S, Kwon H, Hyun C, Ahn S, Choi J, Lee D, Noh SH (2007) Block recycling schemes and their cost-based optimization in NAND flash memory based storage system. In: Proceedings of the 7th ACM international conference on embedded software (EMSOFT'07), Salzburg, Austria, Sep 2007, pp 174–182

26. Kang S, Park S, Jung H, Shim H, Cha J (2009) Performance trade-offs using NVRAM write buffer for flash memory-based storage devices. IEEE Trans Comput 58(6):744–758
27. Lee BC, Ipek E, Mutlu O, Burger D (2009) Architecting phase change memory as a scalable DRAM alternative. In: Proceedings of the 36th international symposium on computer architecture (ISCA), Austin, TX, USA, Jun 2009, pp 1–12
28. Yoon JH, Nam EH, Seong YJ, Kim H, Kim BS, Min SL, Cho Y (2008) Chameleon: a high performance flash/FRAM hybrid solid state disk architecture. IEEE Comput Archit Lett 7(1):17–20
29. Sun K, Baek S, Choi J, Lee D, Noh SH, Min SL (2008) LTFTL: lightweight time-shift flash translation layer for flash memory based embedded storage. In: Proceedings of the 8th ACM international conference on embedded software (EMSOFT'08), Atlanta, Georgia, USA, Oct 2008, pp 51–58
30. Hsieh J-W, Tsai Y-L, Kuo T-W, Lee T-L (2008) Configurable flash-memory management: performance versus overheads. IEEE Trans Comput 57(11):1571–1583
31. Llanos DR TPCC-uva: an open-source implementation of the TPC-C benchmark, installation and user guide, version 1.2. http://www.infor.uva.es/~diego/tpcc-uva.html (2006)
32. Narayanan D, Donnelly A, Rowstron A (2008) Write off-loading: practical power management for enterprise storage. In: Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08), San Jose, CA, USA, Feb 2008, pp 253–267