System-Wide Cooperative Optimization for NAND Flash-Based Mobile Systems

Hyotaek Shim, Jin-Soo Kim, and Seungryoul Maeng

Abstract—NAND flash memory has become an essential storage medium for various mobile devices, but it has some idiosyncrasies, such as out-of-place updates and bulk erase operations, which impair the I/O performance of those devices. In particular, the random write performance is strongly influenced by the overhead of a Flash Translation Layer (FTL) that hides the idiosyncrasies of NAND flash memory. To reduce the FTL overhead, operating systems need to be adapted for FTL, but widely used mobile operating systems still mainly adopt algorithms designed for traditional hard disk drives. Although there have been recent studies on rearranging write patterns into a sequential form in the operating system, these approaches fail to produce sequential write patterns under complicated workloads, and FTL still suffers from significant garbage collection overhead. If the operating system can be made aware of the write patterns that FTL requires, the overhead can be alleviated even under random write workloads. In this paper, we propose a system-wide cooperative optimization scheme, where the operating system communicates with the underlying FTL and generates write patterns that FTL can exploit to reduce the overhead. The proposed scheme was implemented on a real mobile device, and the experimental results show that the proposed scheme constantly improves performance under diverse workloads.

Index Terms—NAND flash memory, flash translation layer, page cache, I/O scheduler, mobile system

1 Introduction

In recent years, mobile computing has rapidly expanded its influence through evolving mobile operating systems and applications. As diverse mobile applications are demanding complicated I/O workloads, the I/O performance has become a critical performance bottleneck for user applications that require more storage capabilities [1]. NAND flash memory [2] has been widely adopted as an essential storage medium in various mobile devices, such as smartphones and tablet PCs. In particular, NAND flash memory has inherent characteristics useful for mobile consumer devices, such as shock resistance, noiselessness, low power consumption, and small form factor.

However, the performance of NAND flash-based storage devices has been impaired by the idiosyncrasies of NAND flash memory, such as the different unit sizes of write and erase operations, and asymmetric read/write latency. The Flash Translation Layer (FTL) that hides and controls such idiosyncrasies is a key component to determine the I/O performance. FTL provides logical-to-physical address mapping for block-oriented file systems by transparently performing garbage collection that copies valid pages before erasing a block. Many recent studies have tried to develop more efficient FTL algorithms that reduce garbage collection overhead and memory consumption [3]–[7].

Improving the operating system (OS) and making the device buffer management policy more advantageous to the underlying FTL have been another interesting research issue. These approaches have also been attempted for magnetic disks: 1) prefetching in the file system and page cache, 2) arranging the file system metadata on each cylinder group, 3) merging several I/O requests into a large sequential request in the I/O scheduler and device buffer, and 4) elevator-based I/O scheduling [8]–[11]. These works focused on reducing seek latency by considering such characteristics of magnetic disks as higher sequential I/O performance, cylinder group size, and seek distance.

In a similar way, there have been many studies on adapting the operating systems or the device write buffer for flash storage devices [12]–[18] by employing the parameters of flash storage devices, such as the flash page/block size and FTL type. Most of the studies have focused on tuning the policies of the device write buffer and I/O scheduler to select large and cold data as a victim in the expectation that evicting large data can generate sequential write patterns, which are favorable to the FTL performance [3], [4]. However, with complicated workloads, the existing approaches fail to produce sequential write patterns. In addition, it is difficult for the previous studies to determine which write requests can minimize the garbage collection overhead as they do not consider the internal state of the underlying FTL. In magnetic disks, only considering current head position can be enough to optimize the performance, whereas the performance of FTLs is influenced by the internal state of FTLs that is too intricate to be tracked.

- *H. Shim and S. Maeng are with the Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Republic of Korea. E-mail: {htshim, maeng}@calab.kaist.ac.kr.*
- *J.-S. Kim is with the School of Information and Communication Engineering, Sungkyunkwan University (SKKU), Suwon, Republic of Korea. E-mail: jinsookim@skku.edu.*

Manuscript received 10 Aug. 2012; revised 13 Feb. 2013; accepted 13 Mar. 2013. Date of publication 26 Mar. 2013; date of current version 15 July 2014. Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.74

0018-9340 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

To break through the limitations of existing studies, this paper presents a system-wide cooperative optimization scheme to enhance the performance of NAND flash-based mobile storage systems under diverse workloads. In the proposed approach, the operating system communicates with the underlying FTL and generates write requests that FTL can utilize to reduce the garbage collection overhead. To the best of our knowledge, we are the first to devise a cooperative approach between the operating system and FTL. Our work is divided into three parts. First, we determine the performance bottleneck of I/O operations in existing hybrid FTLs. Second, we investigate how the bottleneck could be addressed by the cooperative optimization with the operating systems that are capable of inquiring into the internal states of FTLs. We also investigate what information is needed for the operating systems. For the proposed scheme, we are required to define the cooperation interface, which should be low-cost and compatible across different FTLs. Finally, we apply the proposed scheme to a real flash-based mobile system. The experimental results show that the proposed scheme achieves significant improvement in the I/O performance. In the results of the SysBench benchmark, the throughput of the proposed scheme has been improved by 93.3% on average, compared with that of the existing *Android* system.

The remaining sections of this paper are organized as follows. Section 2 contains brief descriptions of NAND flash memory, the overall architecture of a flash-based mobile device, and representative FTL algorithms. Section 3 explains related work. Section 4 gives an explanation of the limitation of the existing studies. Section 5 describes the algorithms of the proposed scheme in detail. Section 6 explains the experimental environment and analyzes evaluation results. Finally, Section 7 summarizes and concludes this paper.

1.1 Characteristics of NAND Flash Memory

NAND flash memory logically consists of an array of erase blocks, each of which contains a fixed number of pages. A page is the unit of read and write operations, and a block is the unit of erase operations. As previously mentioned, NAND flash memory has some idiosyncrasies. Write operations take several times longer than read operations. The write latency is further increased by the erase-before-write feature where write operations can be allowed only on previously-erased pages. The discord between the units of write and erase operations necessitates garbage collection that copies valid pages distributed in old blocks to previously-erased blocks before erasing the old blocks. The amount of garbage collection overhead is a key factor of determining the overall performance of flash-based storage systems.

As another restriction, the number of erase operations on a block is strictly limited from 3,000 to 100,000 [2]. If the erase count of a block exceeds beyond the limit, the block is likely to be worn out and is not recommended to be written any more. Therefore, erase operations need to be evenly distributed over all blocks to achieve wear-leveling. In consideration of this restriction, decreasing the number of write and erase operations is strongly required, not only to enhance the performance, but also to extend the lifetime of flash-based mobile devices.

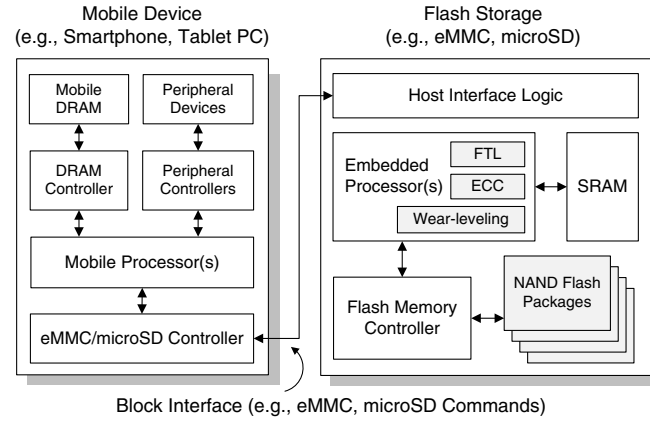


Fig. 1. Overall architecture of a flash-based mobile device.

2 Background

2.1 Architecture of Flash-Based Mobile Devices

Fig. 1 illustrates the overall architecture of a typical flash-based mobile device, which contains one or more mobile processors, mobile DRAM, and flash storage. The applications and operating system are operated in the mobile device, whereas FTL is executed inside the flash storage, controlling address mapping, garbage collection, Error Correction Code (ECC), and wear-leveling.

There are several types of flash storage, such as Embedded Multi-Media Card (eMMC) and Micro Secure Digital Card (microSD) [1]. eMMC is widely used as an internal flash storage, and microSD optionally as an external storage. Such flash storage is connected to the mobile device by using the standard block-oriented interfaces based on the eMMC command set or a variant of it [19], [20]. Through these block interfaces, mobile operating systems can transparently use traditional file systems, such as Ext3 and Ext4.

Commonly in both eMMC and microSD, the internal architecture consists of the embedded processor(s), small-sized SRAM, flash memory controller, and multiple NAND flash packages. The flash memory controller inside the flash storage manages multiple flash packages. Each flash package consists of one or more dies, and each die is composed of one or more planes. A plane contains a number of erase blocks. The operations on different packages can also be interleaved for improving parallelism. In this way, the flash storage augments sequential I/O performance although the bandwidth of only one single plane is strictly confined.

2.2 Flash Translation Layer

The main role of FTL is to make the flash storage emulate one of traditional block devices, hiding the idiosyncrasies of NAND flash memory. An important design trade-off of FTLs is reducing the garbage collection overhead for better performance versus reducing the consumption of memory and computing resources. According to the design trade-off, existing FTLs are categorized into three groups: block-level, page-level, and hybrid FTLs.

The block-level FTLs have the smallest memory consumption, but involve much overhead [21]. Whenever a part of a block is updated, the remaining part is also copied to a new block in company with the updated part. This is because all of the logical pages within a logical block should be located

always on the designated page offset in a physical block due to coarse-grained block-level mapping.

In contrast, the page-level FTLs [5] show the best performance among the three groups, since logical pages can be located on any physical block without restricting the page offset due to fine-grained page-level mapping. However, the applicable systems of page-level FTLs are limited by a large memory footprint.

The hybrid FTLs balance the trade-off between the block-level and page-level FTLs. The hybrid FTLs manage most of the blocks (called data blocks) with block-level mapping, and a small number of blocks (called log blocks) with page-level mapping. Log blocks are used for storing page updates on data blocks. Less frequently-written data are stored in data blocks, whereas more frequently-written data can be updated in log blocks. Based on temporal locality, the performance of hybrid FTLs can approximate that of page-level FTLs with a small amount of memory. The flash storage that is described in the previous subsection usually adopts hybrid FTLs [1] due to the limited memory resources within a controller chip. Therefore, we focus on hybrid FTLs in this paper. Several variants of hybrid FTLs have been developed, which have different associativities between data blocks and log blocks. We briefly explain two representative hybrid FTLs.

2.2.1 Block-Associative Sector Translation (BAST)

The Block-Associative Sector Translation (BAST) FTL [3] adopts the one-to-one association between data blocks and log blocks. A page update on a data block is appended in the corresponding log block. When all the free pages of the log block are consumed, or when the log block is requested for another data block, the log block is merged with its data block. There are three types of merge operations: a switch merge, partial merge, and full merge.

If a log block to be merged was fully and sequentially written with the pages of the same block address, the log block is simply changed to a new data block, and the old data block is erased with all the invalid data. This merge operation is most efficient, and is called a switch merge. If a log block was partly written from the first page in the right offset, the data block can be reclaimed after the remaining free pages of the log block are filled with the valid pages from the data block. This merge is called a partial merge. If a log block was written in a non-sequential manner, all the valid pages distributed in the data block and the associated log block should be sequentially copied into a separate free block, which eventually becomes a new data block. Ultimately, the log block and the old data block are erased. This is called a full merge. Under non-sequential write patterns, the full and partial merges that involve the expensive page copies occupy a major portion of the garbage collection overhead.

2.2.2 Fully-Associative Sector Translation (FAST)

The Fully Associative Sector Translation (FAST) FTL [4] is based on the many-to-many association between data blocks and log blocks. Since log blocks are shared by all data blocks, any update of a data block can be written to any of log blocks called RW log blocks. Because of this policy, RW log blocks are fully utilized, and merge operations can occur less frequently. In addition, if the write pattern exhibits high temporal locality, the log pages of RW log blocks can be repeatedly invalidated, and the merge overhead can be more reduced.

However, sharing all RW log blocks among different data blocks hinders producing lightweight switch merges if there are multiple write threads. To alleviate this problem, FAST stores sequential writes to the separate log block called the SW log block. Nevertheless, sorting out the sequential writes is one of the most challenging issues. FAST roughly distinguishes the sequential writes, as follows. If the first offset of a write request is zero (the first page in a block) and the SW log block is empty, or if the write request is continued from the last page written in the SW log block, the write request is considered as a sequential write and is written into the SW log block. In this way, the SW log block is sequentially written with the pages of the same block address. In other cases, write requests are written into the RW log blocks, which are considered as random writes.

Similar to BAST, there are three types of merge operations. If the SW log block is fully written, it is reclaimed by a switch merge. If the first offset of a write request is zero but the SW log block is not empty, the SW log block is reclaimed by a partial merge. Such misprediction of sequential writes can trigger numerous partial merges. Whenever there are no free log blocks, FAST reclaims one of the written RW log blocks in a round-robin fashion. To reclaim a RW log block, FAST merges all associated data blocks that contain valid pages within the RW log block. To merge the associated data block, FAST copies all the valid pages of the associated data block into a free block. The valid pages may be distributed among several RW log blocks. Thereafter, the fully-written block becomes a new data block, and the associated data block is erased. This full merge is repeated until all the associated data blocks are merged into new data blocks, and the victim RW log block is finally erased. These full merges that occur in bulk have a negative effect on operational latency. Also in FAST, the full and partial merges are major obstacles in improving the performance.

3 Related Works

Recently, several studies have tried to change the write pattern issued from the internal device buffer to alleviate the cost of full merges in hybrid FTLs. The Flash-Aware Buffer (FAB) management scheme [14] was proposed to increase the chance of switch merges by changing the buffer management policy. In FAB, all of the buffer pages that belong to the same logical block are considered as a victim unit, and they are sequentially flushed at the same time. The logical block that contains the largest number of valid pages in the LRU list is selected as a victim block.

Inspired by FAB, sorting out large and cold blocks as victim blocks has received much interest in several studies, such as the Cold and Largest Cluster (CLC) policy [16], the Expectation-based LRU (ExLRU) policy [17], and the Write-Buffer-Cache aware LRU (WBC-LRU) policy [18]. CLC partitions the device buffer into two levels: the size-independent LRU cluster list (block-level LRU) as the upper level and the size-dependent LRU cluster list (FAB-like) as the lower level. From this structure, CLC prevents hot blocks from being evicted due to the large size. ExLRU also uses partitioned

regions (the work region and eviction region) to give large blocks a higher priority of eviction with the consideration of hotness. A victim block is selected from the eviction region by using a heuristic based on several parameters of buffer blocks, such as the average hit count, size, and age. Similarly, WBC-LRU divides buffer blocks into small and large groups to allow the buffer blocks of the small group to get a second chance to grow larger. In particular, WBC-LRU modified the virtual memory of the operating system to preferentially evict a virtual page that is clean or belongs to the buffer blocks in order to enlarge the size of the buffer blocks. In a similar context, the Individual Read Bundled Write FIFO (IRBW-FIFO) [12] tuned the I/O scheduling algorithm to issue all the write requests that belong to the same logical block in bulk.

Block Padding Least Recently Used (BPLRU) [15] is another kind of buffer management scheme based on hybrid FTLs. BPLRU selects a victim block in the block-level LRU order. Before a victim block is evicted, all the omitted pages of the victim block are filled by reading them from FTL. Accordingly, victim blocks are always flushed in the form of complete blocks. BPLRU thoroughly eliminates full merges and always brings switch merges in the underlying FTL. Under sequential write patterns, this scheme can be effective in diminishing the cost of merge operations with few log blocks.

Commonly in all of those studies, they adopt the block-level eviction that batches victim pages within the same logical block and try to select cold and large blocks as a victim in the internal device buffer or in the I/O queue. They expect that evicting those blocks is beneficial in reducing costly full merges in hybrid FTLs. Under sequential write patterns, such an approach could be effective in producing complete block flushes that are likely to trigger lightweight switch merges. However, under complicated write patterns, even such an approach cannot help producing unfavorable write patterns to the hybrid FTLs.

In the next subsection, we explore how effectively the previous approach can perform under complicated I/O patterns. Most of the studies assumed an internal device buffer that has dozens of MB capacity [14]–[18], [22], but typical mobile-based flash storage shown in Fig. 1 has only a small amount of the internal buffer that can hold only just several pages, differently from high-performance storage devices with enough buffers like Solid State Drives (SSDs). Accordingly, those studies based on such large internal buffer cannot be easily applied for the mobile devices equipped with the flash storage. In this environment, we need to pay more attention to OS-level assistance. As a representative example of the preceding studies, we first examine the effectiveness of IRBW-FIFO [12] that aims at reducing merge costs by adapting the I/O scheduler to flash storage devices.

4 Motivation

To evaluate the effectiveness of the block-level batching and eviction in IRBW-FIFO [12], we measure merge costs with the DBench benchmark. The details of the DBench benchmark and the evaluation environment are explained in Section 6, but here we just emphasize that the workload becomes more complicated as the number of threads increases in the benchmark. In this measurement, we run DBench with 4 threads.

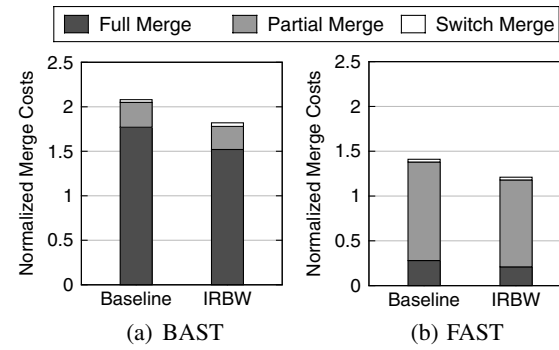


Fig. 2. Breakdown of merge costs normalized to the cost of original operations.

For comparison, we used two different policies: Baseline and IRBW. Baseline means an unmodified Android kernel, and IRBW means IRBW-FIFO that modifies the NOOP-based I/O scheduler, where write requests that belong to the same logical block are batched and sequentially issued in bulk. We did not consider the victim selection policy for the device buffer, because we assumed mobile devices that have no internal device buffer. Fig. 2 shows the breakdown of merge costs in Baseline and IRBW. Each merge cost is normalized to the cost of original operations that exclude merge operations. Compared with Baseline, IRBW slightly reduced full merge costs. However, the summed cost of full and partial merges is more than 120% of the original operation cost, and they are still major performance bottlenecks.

In hybrid FTLs, there is a critical limitation in the perspective of the merge algorithm. If a log block is written at least once in non-sequential order, the log block should involve expensive full merges. In BAST, the randomly-written log blocks are reclaimed by full merges regardless of subsequent write patterns, even with the sequential updates falling into the same block. In FAST, the randomly-written RW log blocks cannot avoid full merges unless they are thoroughly invalidated by following updates. Accordingly, under random write patterns, only batching write requests in the block level is not sufficient to reduce full and partial merges.

For this reason, although the existing approaches reproducing write patterns into sequential form can be somewhat effective in creating more chances for switch merges, they cannot overcome the performance bottlenecks under random write patterns. This limitation of the previous studies inspired us to devise a new approach that can improve I/O performance even under complicated write patterns. Unlike the previous studies based on one-sided optimization in the operating system, our approach is based on the cooperative optimization between the operating system and FTL. In the following section, we investigate how the operating system and FTL can cooperatively work to handle the performance bottlenecks of costly full and partial merges.

5 Cooperative Optimization

To reduce expensive full and partial merges, we propose a system-wide cooperative optimization (COOP) scheme between the operating system and FTL. Based on this cooperation, we devise a novel merge algorithm that replaces

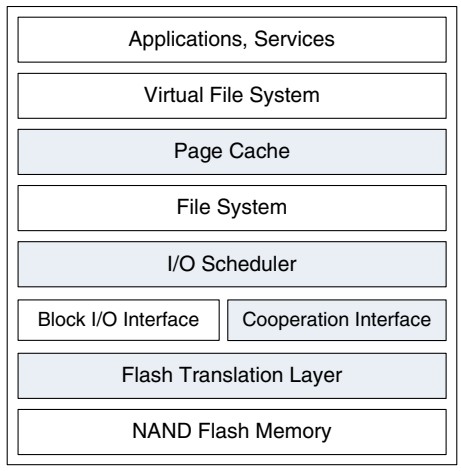


Fig. 3. Overall system hierarchy.

expensive full and partial merges with switch merges in hybrid FTLs. This merge algorithm ensures less overhead, even under random write patterns, which will be explained in the following subsection.

The proposed cooperation is based on the Android mobile platform, which is a representative open-source mobile platform. Fig. 3 represents the overall system hierarchy that includes three major components on which the proposed scheme is implemented: a page cache, I/O scheduler, and FTL module. In the remaining parts, the OS components refer to the page cache and the I/O scheduler.

The OS components adjust write patterns in cooperation with FTL to make FTL avoid expensive merges. The cooperation proceeds briefly as follows. From the I/O scheduler, write requests that belong to the same logical block are issued together in a sequential manner, which we call a block issue. The OS components inform FTL of the parameters of each block issue in advance, and FTL returns the logical block numbers (merge blocks) to be merged by full or partial merges due to the block issue. Through this communication, the OS components give higher eviction priority to the dirty pages that belong to the merge blocks. Ultimately, those preferentially-initiated block issues (called preferential block issues) are exploited by the proposed merge algorithm (called a cooperative switch merge or CSM) in FTL. In the following subsections, we explain detailed algorithms and considerations for each OS component, and the cooperation interface.

5.1 Cooperation Interface between OS and FTL

In practice, the internals of FTLs are quite intricate and also unpredictable since the detailed implementation of FTLs could be different according to the flash storage manufacturers. Moreover, only with the existing block interface, FTL cannot foresee how many pages within the same logical block will be issued, which is indispensable information for the proposed merge algorithm. For the cooperation between the OS components and FTL, we need to define a new interface set that complements the existing block interface.

Table 1 describes the cooperation interface that consists of four commands and their parameters in detail. The flash page/block numbers indicate logical addresses in FTL. First,

TABLE 1
Prototype of the Cooperation Interface

Prototype of Commands
COOP_Query(FPN, nPages) ⇒ MBS
COOP_Issue(FBN)
COOP_Complete(FBN)
COOP_GetSize() ⇒ FBS, FPS

Input/Output Parameters	
FPN	First flash page number of a block issue
nPages	Number of flash pages in a block issue
MBS	Flash block numbers to be merged in FTL
FBN	Flash block number to be issued
FBS, FPS	Flash block size, Flash page size

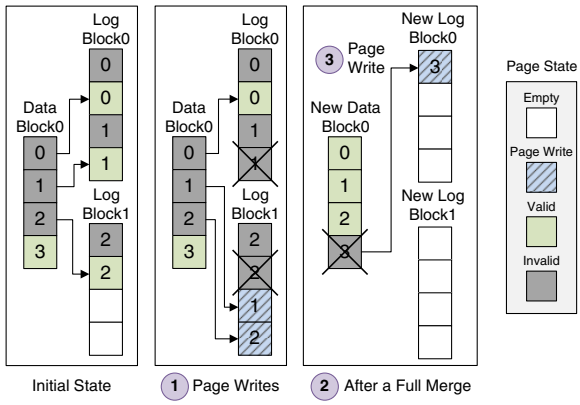
COOP_Query(FPN, nPages) is used for checking whether any full or partial merges will be caused by a block issue to FTL; FPN is the first flash page number of the block issue, and nPages is the number of flash pages. Before initiating block issues, the I/O scheduler delivers the parameters of the block issues to FTL. If any of full or partial merges are anticipated, FTL returns a set of logical block numbers to be merged. COOP_Issue(FBN) and COOP_Complete(FBN) notify FTL of the start and the end of the preferential block issue, respectively. Finally, the logical block/page size of FTL used for the cooperation is obtained by COOP_GetSize(). The detailed algorithms of the cooperation interface are explained in Section 5.4.

As shown in Fig. 3, the cooperation interface is co-located in the same level as the existing block I/O interface. In this paper, we assume the mobile devices equipped with the flash storage, such as eMMC and microSD [1]. Although these flash storage devices use only the standard block interfaces [19], [20], we believe that the proposed scheme could still be applied to those devices by using vendor-specific or obsolete commands. The proposed interface is simple and generic enough to be combined with the existing interface, and can be compatibly used by the OS components for different hybrid FTLs, such as BAST and FAST.

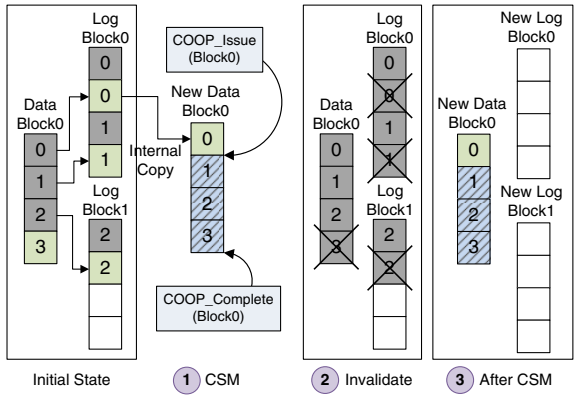
5.2 Cooperative Switch Merge

As a key algorithm of the cooperative optimization, we devise an advanced merge algorithm that prevents the full or partial merge in hybrid FTLs, which is called a Cooperative Switch Merge (CSM). In CSM, if full or partial merges are anticipated in a block where a block issue arrives, the block issue is written fully in a separate free block as FTL internally copies the omitted pages of the issue. The fully-written block becomes a new data block, and then the old pages in log and data blocks are all invalidated. In this way, the old log and data blocks can be easily reclaimed without full or partial merges. If full or partial merges are not anticipated, block issues are written in normal log blocks without additional overhead in order to minimize the cost of internal copies.

Fig. 4 depicts the procedures of handling an incoming block issue (Page1∼3) with and without CSM in the FAST FTL. In the example of Fig. 4(a) without CSM, the front pages of the block issue (Page1∼2) are first written in the remaining free pages in the Log Block1 (Fig. 4(a)①). Since there are no more free pages in the log blocks (Log Block0∼1), some of the log blocks should be reclaimed to accommodate the remaining page write of the block issue (Page3). Since the log blocks



(a) Without CSM



(b) With CSM

Fig. 4. Procedures of handling a block issue (Page1∼3) with/without CSM in the FAST FTL.

were written in a non-sequential manner, a costly full merge is inevitable. After the full merge is finished (Fig. 4(a)②), the last page of the block issue (Page3) is written in the newly-allocated log block, invalidating the new data block again (Fig. 4(a)③).

However, as shown in Fig. 4(b), the full merge can be replaced with the lightweight CSM. If an incoming block issue corresponds to the merge blocks, FTL can utilize the block issue (Page1∼3) instead of valid page copies while reclaiming merge blocks. FTL redirects the block issue to a separate free block as internally copying the omitted page (Page0), and creates a new data block (Fig. 4(b)①). The start and end of a block issue on the merge block (Block0) are notified by the *COOP_Issue* and *COOP_Complete* commands, respectively. The fully-invalidated Data Block0 and Log Block0∼1 can be simply erased (Fig. 4(b)②). In this way, the expensive full merge is prevented, and the two empty log blocks are obtained (Fig. 4(b)③).

The advantages of CSM are threefold. First, the expensive merges that involve numerous valid page copies are prevented. CSM can reclaim log and data blocks with less overhead, even under random write patterns. Second, block reclamation and block issues can occur concurrently, so that the write latency can be reduced. Without CSM, handling block issues should be delayed until costly merges are finished, in which case the write latency is significantly increased. Third, reclaimed log blocks remain completely empty, and following merge operations are further delayed. In the existing hybrid FTLs, the log blocks always remain partly-written after receiving a block issue that involves full or partial merge operations.

Algorithm 1 Cooperative Switch Merge

```
1:  //FBN: Flash block number
2:  //RPN: Requested logical page number
3:  //CSM_Log: A log block used for CSM
4:  procedure COOP_ISSUE(FBN)
5:      if CSM_Log is not allocated then
6:          Allocate CSM_Log for the FBN data block;
7:      end if
8:  end procedure
9:
10: procedure FTL_WRITE(RPN)
11:     if Block # of RPN = FBN then
12:         RPO ← Page offset of RPN in the block;
13:         NPO ← Next free page offset in CSM_Log;
14:         while NPO < RPO do
15:             Internal valid copy to the NPO log page;
16:             NPO++;
17:         end while
18:         Write the RPN page to the NPO log page;
19:         NPO++;
20:     else
21:         Write the RPN page in the existing ways;
22:     end if
23: end procedure
24:
25: procedure COOP_COMPLETE(FBN)
26:     while NPO < # of pages per block do
27:         Internal valid copy to the NPO log page;
28:         NPO++;
29:     end while
30:     Change CSM_Log to a new data block;
31:     Erase the old data and log blocks;
32: end procedure
```

Algorithm 1 describes the detailed algorithm of *COOP_Issue*, *FTL_Write*, and *COOP_Complete* routines used in CSM, in which the page/block number indicates the flash page/block number in FTL. The word in parenthesis means the input of the command, and this notation is also used in the following algorithms. Before carrying out CSM, FTL receives the flash block number (FBN) from the *COOP_Issue* command, and then a separate log block (CSM_Log) is allocated. After this time, all the write requests of the block issue on FBN are redirected to CSM_Log in the right page offset. Other write requests beyond FBN are written in normal log blocks and are

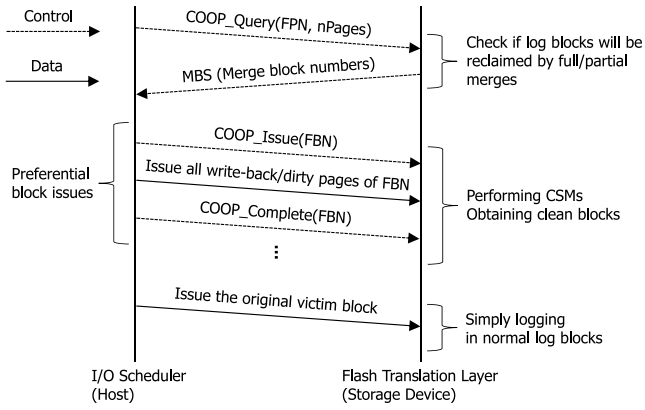


Fig. 5. Communications for the cooperation between the I/O Scheduler and FTL.

handled in the existing ways. The omitted pages between write requests are filled by internal copies in FTL, and can be detected by comparing the next free page offset of CSM_Log and the first page offset of each write. When the block issue on FBN is finished, which is notified by the *COOP_Complete* command, the remaining omitted pages are internally copied and filled if exist. Finally, the fully-written CSM_Log becomes a new data block, and the invalidated old data and log blocks can be simply erased.

An important requirement to make the best use of the lightweight CSM is that the log blocks to be reclaimed by full or partial merges should be given priority of receiving block issues. For this requirement, FTL should foresee the parameters of block issues and adjust the order of block issues from the OS components by using the cooperation interface defined in Table 1.

5.3 Cooperation of the OS Components

5.3.1 Block-Level Flush and Issue

In the Linux kernel used in the Android system, the page cache and I/O scheduler adopt the inode-level flush and request-level issue policies, respectively. In the page cache, when an inode is synchronized, all the dirty pages within the inode are flushed. The dirty pages flushed from the page cache are temporarily queued in the I/O scheduler in the form of multiple write requests each of which represents continuous logical pages in FTL. These policies themselves can be advantageous to hybrid FTLs when large files are sequentially written because the I/O scheduler usually merges adjacent requests into a large sequential request. However, with many small files or with randomly-written large files, these policies inevitably generate random write patterns according to the order of synchronizing inodes in spite of the inherent request-merging policy in the I/O scheduler.

To alleviate the complexity of write patterns under random writes, the block-level flush and issue polices need to be adopted in the page cache and I/O scheduler, respectively. There are three reasons why the proposed scheme uses the block-level flush/issue (eviction) policies in the OS components for hybrid FTLs. First, the block-level eviction can create more chances to make switch merges, as mentioned in Section 4. Second, the block-level eviction can increase log block utilization in the BAST FTL, and decrease merge latency by reducing the number of associated blocks within a RW log block in the FAST FTL. Third, the proposed CSM necessitates block-level eviction to anticipate full and partial merges, and to reduce the amount of internal copies.

In the existing page cache, when an inode is synchronized, the dirty pages are flushed in file index order regardless of the location in the storage system. In the cooperative page cache, the dirty pages of the same block within the inode are flushed together as the flush unit; the flash block size can be obtained by the *COOP_GetSize* command. To avoid frequent page flushes, the flush unit excludes the other dirty pages of the same block from different inodes.

In the existing I/O scheduler (e.g., NOOP), a victim request is selected among read and write requests in the FIFO order. Likewise, in the cooperative I/O scheduler, the oldest request is selected as a victim in the I/O queue. If the victim is a read request, it is simply issued without being batched. Otherwise, all the write requests within the same logical block of the victim request are batched into a victim block, and they are issued together in a sequential manner.

5.3.2 Cooperative Optimization

Based upon the block-level eviction in the page cache and I/O scheduler, both of the OS components cooperatively

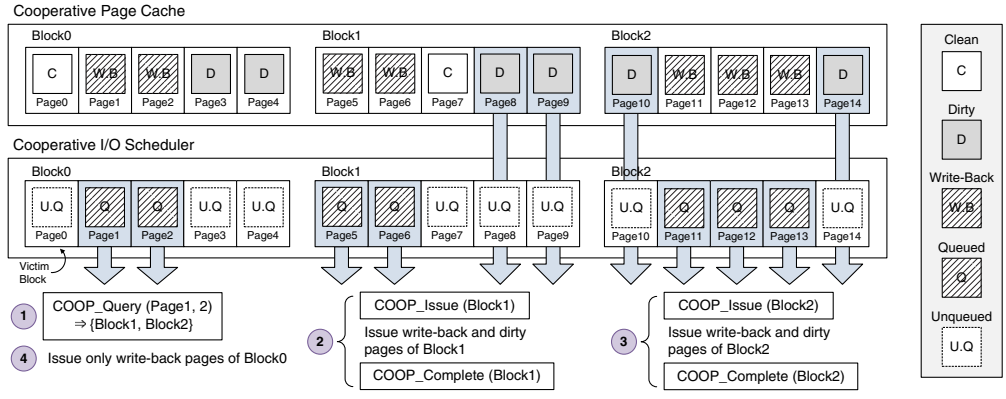


Fig. 6. Operations of the OS components: the page cache and I/O scheduler.

work with FTL to meet the requirement of CSM as explained in Section 5.2. In the proposed scheme, the I/O scheduler communicates with FTL by using the cooperation interface as the final stage before the flushed dirty pages reach the flash storage. Fig. 5 illustrates the communication procedure for the cooperation between the I/O scheduler and FTL. The communication procedure is comprised of giving prior notice to FTL about the parameters of a block issue and preferentially issuing merge blocks according to the response from FTL. The adjusted order of block issues from the I/O scheduler helps FTL to utilize the proposed CSM. Finally, the original victim block is issued from the I/O scheduler, which can be simply appended in normal log blocks without merge operations in FTL.

Fig. 6 depicts an example of performing block issues in the cooperative page cache and I/O scheduler, based on the communication procedure explained in Fig. 5. Note that there are five kinds of page states: clean, dirty, write-back, queued, and unqueued. The write-back state represents that the dirty pages are queued and being waited to be issued in the I/O queue. After a page is synchronized, the page state is changed from write-back to clean. As shown in Fig. 6①, Block0 is selected as a victim, and the I/O scheduler makes a query to FTL with the first flash page number (Page1) and the number of flash pages (2). Since two merge blocks (Block1∼2) are indicated by FTL, the I/O scheduler preferentially performs two block issues (Block1∼2) with notices at the start and end of each block issue (Fig. 6②–③). The cost of the preferential block issues is offset because the page writes substitute for valid page copies while CSM is performed. Therefore, those block issues do not need to be limited within the write-back pages in the I/O scheduler, but include all the dirty pages that stay in the page cache. We call this the *issue range extension*. The preferential block issues (Block1∼2) include the dirty pages in the page cache (Page8, Page9 and Page10, Page14, respectively). In this way, the state of these dirty pages is changed to clean, and the page cache can reduce the following synchronization overhead by flushing the dirty pages that belong to the merge blocks in advance. Finally, only the write-back pages of the original victim block (Block0) are issued without any notice (Fig. 6④).

The cooperative OS components usually evict write-back and dirty pages in the block level, while enhance the eviction priority and provide the issue range extension for the merge blocks. This cooperation of the OS components creates many opportunities for FTL to perform CSMs with little overhead, and reduces the cost of synchronizing inodes.

Algorithm 2 describes the detailed algorithm of the cooperative I/O scheduler. Before issuing a write request (Req), the I/O scheduler makes a query to FTL by using the *COOP_Query* command with the parameters of the block issue: the first flash page number (FPN) and the number of flash pages (nPages). By using the parameters, FTL can anticipate whether any full or partial merges will occur when the victim block is issued. The detailed algorithms of the anticipation are explained in Section 5.4. If FTL returns the merge block numbers (MBS), the I/O scheduler preferentially initiates the block issues that belong to the merge blocks. As explained in the above section, the preferential block issues (MPS) include all the write-back (MQPS) and dirty pages (MDPS) of the merge blocks, and are handled by lightweight CSMs instead of expensive full and partial merges in FTL. The CSMs are conducted with the *COOP_Issue*, *FTL_Write*, and *COOP_Complete* commands. The algorithms of these commands were explained in Algorithm 1. Ultimately, the I/O scheduler issues the original request (Req) unless the block number of the request was contained in the merge blocks. If there is no dirty and write-back page of the merge blocks in the page cache and I/O queue, the original victim request is handled in the existing ways without preferential block issues.

Algorithm 2 Cooperative I/O Scheduler

```
1:  //Req: A request to be issued from the I/O queue
2:  //MBS: Set of block numbers to be merged in FTL
3:  procedure ISSUE_REQUEST(Req)
4:      if Req type is read then
5:          Read request is handled in the existing ways;
6:      else if Req type is write then
7:          FPN ← First flash page # of Req;
8:          nPages ← # of flash pages in Req;
9:          MBS ← COOP_Query(FPN, nPages);
10:         while ∃ MBN_j ∈ MBS do
11:             MQPS ← Set of the write-back pages of MBN_j in the I/O queue;
12:             MDPS ← Set of the dirty pages of MBN_j in the page cache;
13:             MPS ← MQPS ∪ MDPS;
14:             if MPS is not empty then
15:                 COOP_Issue(MBN_j);
16:                 FTL_Write(MPS);
17:                 COOP_Complete(MBN_j);
18:             end if
19:         end while
20:         if Block # of Req ∉ MBS then
21:             FTL_Write(Write-back pages of Req);
22:         end if
23:     end if
24: end procedure
```

5.4 Cooperation of the Hybrid FTLs

In the proposed scheme, FTL takes a vital role in the cooperation. FTL provides the cooperation interface for the operating system, and is also in charge of determining the order of block issues for CSM according to its current internal state. In this section, we explain the implementation of the cooperation interface in the two representative hybrid FTLs shown in Table 1.

Algorithm 3 COOP_Query Command for BAST

```
1:  procedure COOP_QUERY(FPN, nPages)
2:      FBN ← Logical block # of FPN;
3:      Log ← Log block of FBN;
4:      nRFP ← # of the remaining free pages of Log;
5:      if Log was allocated then
6:          if nPages < nRFP then
7:              MBS ← ∅;
8:          else if Log was sequentially-written and the
    block issue fits into the free pages then
9:              MBS ← ∅;
10:         else if nPages > nRFP then
11:             MBS ← FBN;
12:         end if
13:     else if There are no free log blocks then
14:         MBS ← Logical block #s of victim log blocks;
15:     end if
16:     return MBS;
17: end procedure
```

Algorithm 4 COOP_Query Command for FAST

```
1:  //CSM_TH: Threshold of whether to trigger CSM
2:  procedure COOP_QUERY(FPN, nPages)
3:      FBN ← Logical block # of FPN;
4:      if nPages ≤ CSM_TH then
5:          if nPages < # of remaining free pages then
6:              MBS ← ∅;
7:          else
8:              MBS ← Set of associated block numbers of the
    victim RW log blocks;
9:          end if
10:     else
11:         MBS ← FBN;
12:     end if
13:     return MBS;
14: end procedure
```

Algorithm 3 and 4 describe the detailed algorithms of the *COOP_Query* command in BAST and FAST, respectively. The *COOP_Issue* and *COOP_Complete* commands were previously explained in Algorithm 1, and the *COOP_GetSize* command can be straightforwardly implemented both in BAST and FAST. In Algorithm 3 and Algorithm 4, the page/block number indicates the flash page/block number in FTL, and the input/output parameters of the *COOP_Query* command were previously described in Table 1.

The *COOP_Query* interface is implemented in BAST as follows. For a given block issue (FPN, nPages), if the number of the remaining free pages of the corresponding log block (nRFP) is more than the number of pages to be issued ($\mathrm{nPages} < \mathrm{nRFP}$), no merge operation is expected. If the block issue completely fits into the remaining free pages in the right offset, which can be identified by the parameters of the block issue, a switch merge will occur. For those two cases, FTL does not need to return merge blocks ($\mathrm{MBS} \leftarrow \emptyset$) without incurring any overhead.

If the free pages of the corresponding log block (nRFP) are insufficient to store the block issue ($\mathrm{nPages} > \mathrm{nRFP}$), a full merge is anticipated on the log block ($\mathrm{MBS} \leftarrow \mathrm{FBN}$). Particularly in this case, the merge overhead is significantly increased without CSM, because the full merge is triggered after a part of the block issue is written to the remaining free pages, and then the remaining part of the block issue partly occupies the new log block after the reclamation. Finally, if the log block is not allocated and there is no free log block, some of the pre-allocated log blocks should be reclaimed ($\mathrm{MBS} \leftarrow$ Logical block #s of the victim log blocks). In those two cases, FTL returns the logical block numbers to be merged (MBS), on which FTL requires to enhance the eviction priority in the OS components, and CSM can be performed.

In FAST, if the free pages in the RW log blocks are enough to accommodate the block issue ($\mathrm{nPages} <$ # of remaining free pages), FTL does not return any merge blocks ($\mathrm{MBS} \leftarrow \emptyset$), and then the block issue is simply appended. If the free pages are not enough, FTL returns the associated block numbers of the full merges that are anticipated when reclaiming the victim RW log blocks ($\mathrm{MBS} \leftarrow$ Set of associated block numbers of the victim RW log blocks).

In addition, through the cooperation interface, FAST can adopt an efficient policy of sorting out sequential writes. Since FTL can foresee the exact number of pages for each block issue (nPages), if the number is larger than the predefined threshold ($\mathrm{nPages} > \mathrm{CSM\_TH}$), FTL considers the block issue as sequential writes and handles the block issue with CSM. In this way, frequent partial merges caused by misprediction of sequential writes can be considerably reduced.

6 Evaluation

6.1 Implementation

The proposed scheme was implemented in the real mobile experimental device (*ODROID-T*) [23] based on the Android 2.2 platform (*Froyo*) running the Linux kernel 2.6.32.9. The device contains ARM Cortex-A8 CPU, 512 MB Mobile DDR SDRAM, 512 MB OneNAND flash memory, and the external MicroSD card. To implement the proposed scheme, we modified the Linux kernel including the page cache and I/O scheduler. As it is not supported to modify the FTL firmware in a real eMMC chip, eMMC is emulated by loading the modified FTL modules as device drivers on top of the OneNAND flash memory. Storage benchmarks were executed on this emulated flash storage, while the Android system was operated separately on the MicroSD card without interfering with the emulated storage.

In the OneNAND flash memory, the page size is 4KB and the block size is 256KB. The portion of log blocks was

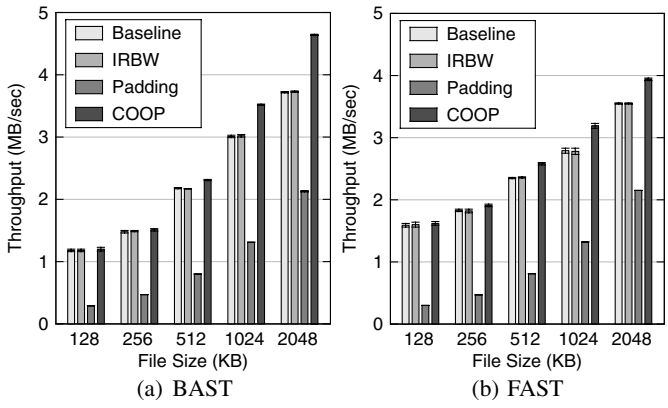


Fig. 7. Comparison of I/O throughputs in the PostMark benchmark.

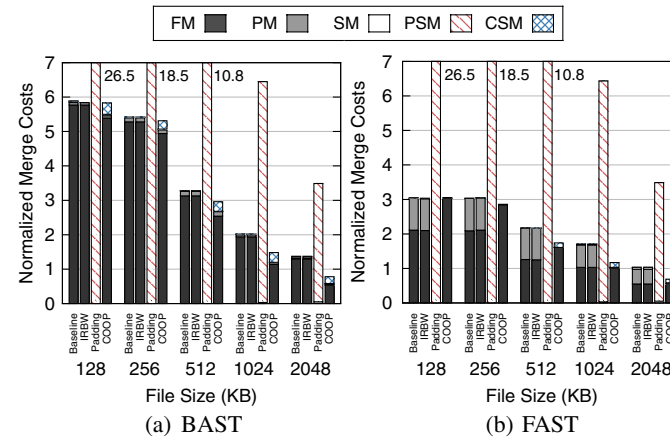


Fig. 8. Breakdown of merge costs in the PostMark benchmark.

configured to 5% of the total capacity. In the hybrid FTLs including BAST and FAST, victim log blocks are selected in the LRU order. In FAST, we configured the threshold (CSM_TH) as half the number of pages in a block. The proposed cooperation interface in Table 1 was implemented in the Memory Technology Device (MTD) layer.

The Linux kernel of the Android system has some disk-based I/O schedulers, such as NOOP, Deadline, and Completely Fair Queuing (CFQ). We selected NOOP as the default I/O scheduler because NOOP is known to be appropriate for the NAND flash storage [13], [24]. The NOOP scheduler merges adjacent requests in the I/O queue and evicts victim requests simply in the FIFO order. For convenience of implementing the block-level eviction in the I/O scheduler, we modified the request merge policy to merge write requests within the flash block boundary, and the maximum size of a request was modified to the flash block size.

In the cooperative page cache, the global cache tree was devised for efficiently searching all of the dirty pages in the page cache, and was implemented by using the radix tree structure for fast indexing. The memory and computing overhead of the global cache tree were reflected in the overall performance. For the convenience of implementing the global cache tree, delayed allocation was disabled in the Ext4 file system.

6.2 Evaluation Methodology

The performance of the proposed scheme was compared with the unmodified Linux kernel (Baseline), the block-level eviction policy (IRBW) [12], and the page padding policy (Padding) [15]. The block-level eviction policy is widely used in recent studies [12], [14]–[18] on designing the I/O scheduler and device buffer management policy, as explained in Section 4. Representatively, the IRBW policy adopts the block-level eviction of write requests in the I/O scheduler. Enhancing the eviction priority of read requests (IRBW-FIFO-RP) was not considered because it is orthogonal with the proposed scheme, and the benchmarks generate write-intensive workloads. The Padding policy additionally applies page padding [15] for IRBW. Based upon the different policies, we evaluated the performance of the proposed scheme with four widely-used storage benchmarks: PostMark [25], SysBench [26], FileBench [27], and DBench [28], which offer diverse workload scenarios.

6.3 PostMark

This experiment compares the performance of each policy under small random writes. Especially, this workload is known to be quite painful for hybrid FTLs. The execution of the PostMark benchmark is divided into three phases: creation, transaction, and deletion. During the creation phase, the benchmark creates a fixed number of files from 8 to 128 in the root directory, varying the file size from 128KB to 2 MB. For each transaction during the transaction phase, a new file is created, or a randomly-selected file is deleted or appended with the randomly-selected amount of data. As the file size is small, the amount of appended data for each transaction is also small. After each append operation, the file is synchronized to generate random write patterns. The number of transactions is configured to 1536. Lastly, during the deletion phase, all of the created files are deleted. We executed the PostMark benchmark four times for each configuration with BAST and FAST. The first run is used for aging the flash storage, and the experimental results are taken from the average value of the other runs.

Fig. 7 shows the average throughput (MB/sec) and its standard deviation, varying the file size under the different schemes. To find out the reason for performance improvement, we also obtained the cost of merge operations during the same execution, as shown in Fig. 8. There are five types of merge operations, such as full merge (FM), partial merge (PM), switch merge (SM), padded switch merge (PSM), and the proposed CSM. The cost of PSM includes the padding overhead, and the cost of CSM reflects the overhead of internal copies. Each merge cost of Fig. 8 is normalized to the overall cost of the original operations.

The results of Baseline and IRBW are almost the same because these policies bring similar access patterns to FTL due to the synchronous append operations. In this workload, the block-level eviction of IRBW is ineffective to reduce full and partial merges, and Padding rather considerably impairs the throughput due to the large padding overhead as shown in Fig 8. Compared with Baseline, the throughput of COOP is improved by 10.4% (BAST) and 8.3% (FAST) on average,

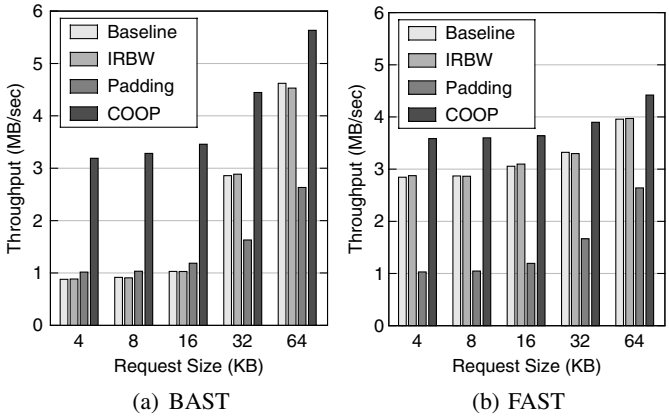


Fig. 9. Comparison of I/O throughputs in the SysBench benchmark.

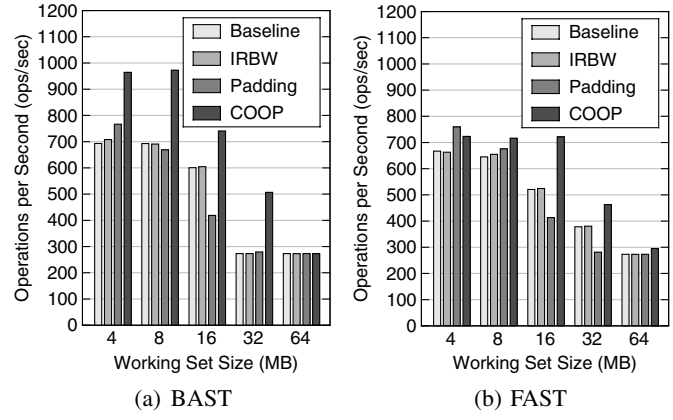


Fig. 11. Comparison of operations per second in the FileBench benchmark.

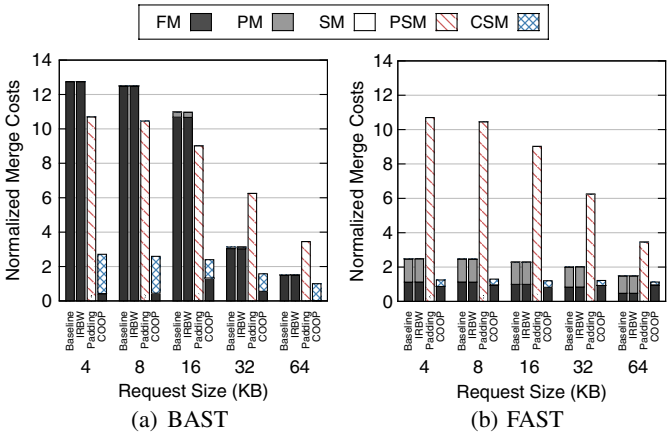


Fig. 10. Breakdown of merge costs in the SysBench benchmark.

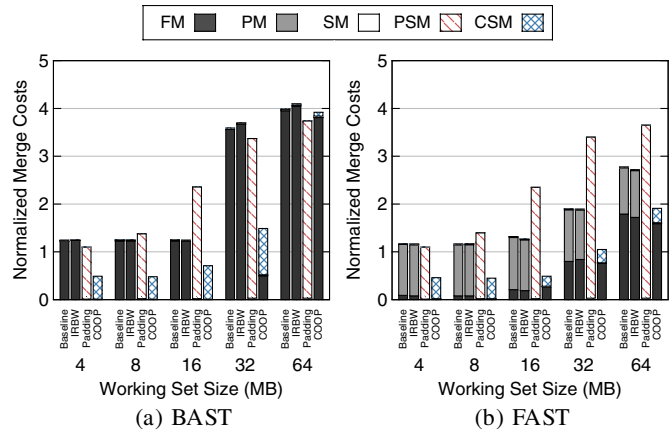


Fig. 12. Breakdown of merge costs in the FileBench benchmark.

and by 24.6% (BAST) and 14.6% (FAST) in the best case. The performance enhancement that the proposed scheme achieved indicates that the cooperation can be effective in reducing full and partial merges, even under such severe small random writes. As the file size increases, the proposed scheme achieves better performance because the overhead of internal copies can be more diminished with the large request size.

6.4 SysBench

In this evaluation with the SysBench benchmark, we compare the performance of each policy under asynchronous random writes. This benchmark creates a large file (32 MB) and writes data into a random position with the predefined request size. The file is synchronized every time the amount of written data becomes 4 MB. The benchmark runs 100000 requests after aging the flash storage for 30 seconds, and the maximum execution time is configured to 180 seconds.

Figs. 9 and 10 present the throughput (MB/sec) and the breakdown of merge operations, respectively, varying the request size from 4KB to 64KB. Also in this workload, Baseline and IRBW show similar throughputs. Although the performance of Padding is somewhat increased than that of Padding in the PostMark benchmark due to the large-sized block issues, the padding overhead is still several times larger than the original operation cost. The throughput of COOP is increased more than that of Baseline by 166.7% (BAST) and 19.9% (FAST) on average, and by 262.5% (BAST) and 26% (FAST) in the best case.

Especially, the reason why the throughput is significantly raised in BAST is that large and random write patterns create many cases where the free pages of the corresponding log blocks are insufficient to accommodate block issues. These cases would involve expensive full merges without CSM, but the proposed scheme can have more opportunities to reduce the merge overhead by replacing these cases with efficient CSMs as shown in Fig. 10(a). In all the evaluation results, the proposed scheme achieves the lowest amount of merge costs.

6.5 FileBench

In this evaluation, we further investigate the random write performance of the proposed scheme with different working set sizes. The FileBench benchmark creates a 256 MB file, and performs 8KB asynchronous random write operations for 60 seconds for each run. The file is synchronized whenever the amount of written data becomes the predefined working set size. We ran the benchmark six times, and obtained the average results for the last five times.

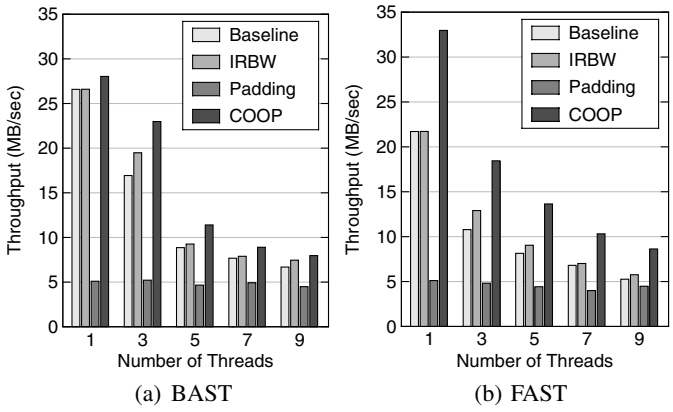


Fig. 13. Comparison of I/O throughputs in the DBench benchmark.

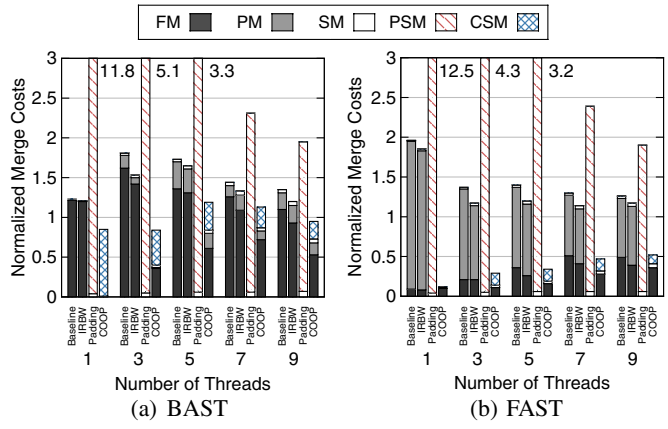


Fig. 14. Breakdown of merge costs in the DBench benchmark.

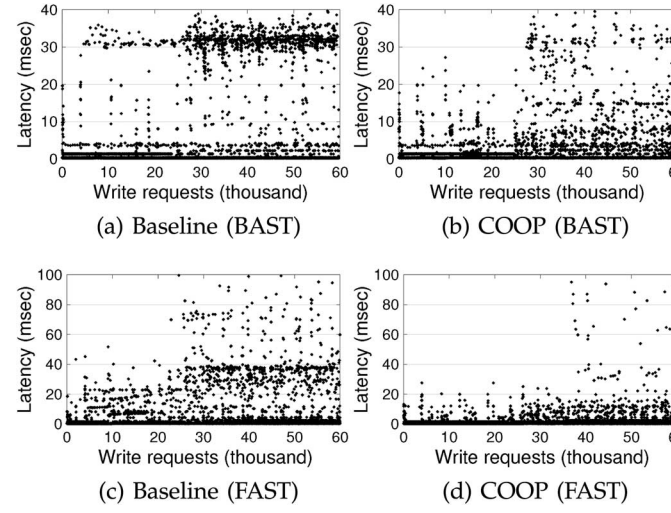


Fig. 15. Comparison of write latency in the DBench benchmark.

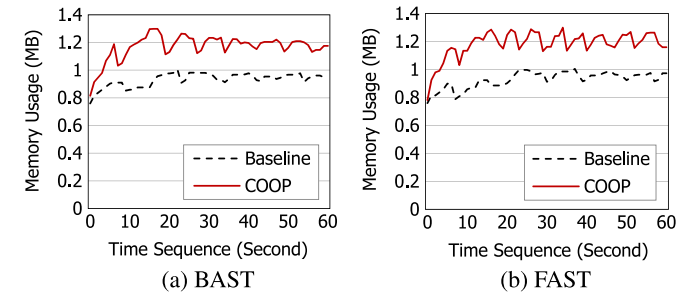


Fig. 16. Memory overhead used for maintaining the global cache tree in the DBench benchmark.

Figs. 11 and 12 show the operations per second (OPS) and the breakdown of merge costs, varying the working set size from 4 MB to 64 MB. The OPS of COOP is larger than that of Baseline by 37.8% (BAST) and 17.7% on average. In the result of Fig. 11, there is a small performance improvement with the proposed scheme when the working set size is 64 MB, which is larger than the total amount of log blocks (25.8 MB). In this situation, victim log blocks exhibit a large working set size, and have few write-back and dirty pages only with the single file regularly synchronized, so that there are small chances of performing CSMs as shown in Fig. 12(a). From this evaluation, we confirm that the proposed scheme can enhance the I/O performance with sufficient log blocks even under random write workloads.

6.6 DBench

In this evaluation with the DBench benchmark, we measure the performance of each policy under complicated workloads with mixed sequential and random writes. The DBench benchmark is used to stress a storage system with a different number of threads, each of which performs file creation, deletion, renaming, reading, writing, and flushing out data. The benchmark is performed for 72 seconds, including the aging period for 12 seconds, with the different number of benchmark threads. As the number of threads increases, the amount of requested data increases, and the access pattern becomes more complicated.

Figs. 13 and 14 present the throughput (MB/sec) and the breakdown of merge costs, respectively, varying the number of threads from one to nine. In this workload, the block-level eviction of IRBW is less effective in reducing the expensive full and partial merges, and Padding raises the merge cost because of the padding overhead. The throughput of COOP is increased more than that of Baseline by 21% (BAST) and 61.2% (FAST) on average. By using the cooperation, a large portion of full and partial merges are redirected to the efficient CSMs. Especially, in the results with FAST, COOP eliminated all of the partial merges because sequential writes were effectively sorted out by the proposed cooperation. The performance results indicate that the proposed scheme can achieve the best performance with complicated workloads.

Fig. 15 illustrates the latency of write operations when the number of threads is four, and each point means the latency of a single page write operation. The average latencies (msec) of Baseline and COOP are 1.02 and 0.6 in BAST, and 1.02 and 0.56 in FAST, respectively. Compared with Baseline, the proposed scheme removed a number of significantly-delayed writes, since alleviating the expensive merge cost leads to the smaller write latency. As explained in Section 5.2, in the existing system, write operations that involve the expensive merges should wait until the merges are completed, whereas the

cooperation scheme can concurrently handle write and merge operations through CSM, reducing the write latency. From this achievement, we envision that the proposed scheme can provide a short response time for interactive user applications in NAND flash-based mobile devices.

To examine the memory overhead needed for maintaining the global cache tree, we measured the memory size allocated for radix tree nodes in the Android kernel with the DBench benchmark. Fig. 16 shows the variation of the memory usage when the number of threads is five. In the COOP scheme, the memory usage is increased by 26.5% on average, compared with that of Baseline, but the amount of the increased memory usage is negligible compared to the available memory size.

7 Conclusion

In this paper, we present a system-wide cooperative optimization scheme to enhance the performance of NAND flash-based mobile storage systems. We apply the proposed scheme to the three major components of the real mobile system, such as the page cache, I/O scheduler, and FTL. For each component, we devise a general cooperative algorithm. In particular, we propose the cooperative switch merge (CSM) as an efficient merge algorithm, which substitutes for costly full and partial merges in hybrid FTLs. For the cooperation, we also suggest the cooperation interface that is compatible across different hybrid FTLs.

In the evaluation with widely-used benchmarks, such as PostMark, SysBench, FileBench and DBench, the proposed scheme results in a significant performance enhancement and also reduces the write latency, compared with several recent studies. In particular, in the results of the SysBench benchmark, the throughput of the proposed scheme has been improved than that of the existing Android system by 93.3% on average.

Based upon the evaluation results, we demonstrate that the proposed system-wide cooperation can alleviate the performance bottleneck (full and partial merges) under diverse write patterns. Although we assumed that the cooperation interface can be combined with the standard block interface by exploiting vendor-specific commands, we believe the performance enhancement that we achieved provides a reasonable basis for manufacturers to consider publicly adopting the proposed interface into the standard interfaces for the flash storage.

Acknowledgment

This research was supported by the SW Computing R&D Program of KEIT (2012-10041313, UX-oriented Mobile SW Platform) funded by the Ministry of Knowledge Economy.

References

[1] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST'12)*, San Jose, CA, Feb. 2012, pp. 209–222.
[2] *1G x 8 Bit/2G x 8 Bit/4G x 8 Bit NAND Flash Memory (K9XXG08UXM) Data Sheets*, Samsung Electronics, Nov. 2005.
[3] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[4] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, article 18, Jul. 2007.
[5] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw.—Pract. Exp.*, vol. 29, no. 3, pp. 267–290, Mar. 1999.
[6] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proc. 6th ACM Int. Conf. Embedded Softw. (EMSOFT'06)*, Seoul, Republic of Korea, Oct. 2006, pp. 161–170.
[7] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Architectural Support Program. Languages Oper. Syst. (ASPLOS'09)*, Washington, DC, Mar. 2009, pp. 229–240.
[8] E. Shriver, C. Small, and K. A. Smith, "Why does file system prefetching work?" in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC'99)*, Monterey, CA, Jun. 1999, pp. 71–84.
[9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 181–197, 1984.
[10] X. Ding, S. Jiang, and F. Chen, "A buffer cache management scheme exploiting both temporal and spatial localities," *ACM Trans. Storage*, vol. 3, no. 2, article 5, Jun. 2007.
[11] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *Proc. Winter USENIX Tech. Conf.*, Washington, DC, Jan. 1990, pp. 313–323.
[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drives," in *Proc. 9th ACM Int. Conf. Embedded Softw. (EMSOFT'09)*, Grenoble, France, Oct. 2009, pp. 295–304.
[13] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Trans.Comput.*, vol. 61, no. 5, pp. 636–649, May 2012.
[14] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-aware buffer management policy for portable media players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.
[15] H. Kim and S. Ahn, "A buffer management scheme for improving random writes in flash storage," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST'08)*, San Jose, CA, Feb. 2008, pp. 239–252.
[16] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Trans. Comput.*, vol. 58, no. 6, pp. 744–758, Jun. 2009.
[17] L. Shi, J. Li, C. j. Xue, C. Yang, and X. Zhou, "ExLRU: A unified write buffer cache management for flash memory," in *Proc. 11th ACM Int. Conf. Embedded Softw. (EMSOFT'11)*, Taipei, Taiwan, Oct. 2011, pp. 339–348.
[18] L. Shi, C. J. Xue, and X. Zhou, "Cooperating write buffer cache and virtual memory management for flash memory based systems," in *Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS'11)*, Chicago, IL, Apr. 2011, pp. 147–156.
[19] *Embedded Multi-Media Card (eMMC), Electrical Standard (4.5 Device)*, JEDEC Standard, Jun. 2011.
[20] SD Group, "SD specifications part 1 physical layer simplified specification version 3.01," SD Card Association, May 2010.
[21] A. Ban, "Flash file system," U.S. Patent 5,404,485, Apr. 1995.
[22] H. Shim, D. Jung, J. Kim, J.-S. Kim, and S. Maeng, "Co-optimization of buffer layer and FTL in high-performance flash-based storage systems," *Des. Autom. Embedded Syst.*, vol. 14, no. 4, pp. 415–443, Dec. 2010.
[23] ODROID-T Platform Developer Edition, Hardkernel Co., Ltd., May 2011.
[24] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST'12)*, San Jose, CA, Feb. 2012, pp. 155–169.
[25] J. Katcher, "PostMark: A new file system benchmark," Report of Network Appliance, Tech Rep. TR3022, 1997.
[26] A. Kopytov, "SysBench: A system performance benchmark," 2004.
[27] FileBench Version 1.4.9.1, "A file system and storage benchmark," Jul. 2008.
[28] A. Tridgell and R. Sahlberg, "DBENCH: A benchmark tool," 2008.

Hyotaek Shim received the BS degree in computer engineering from Inha University, Incheon, Republic of Korea, in 2006, and received the MS–PhD joint degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 2013. He is currently a senior research engineer at Samsung Electronics. His research areas include flash memory-based storage systems, embedded systems, operating systems, and cloud computing systems.

Jin-Soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor at Sungkyunkwan University, Suwon, Republic of Korea. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received the BS degree in electronics engineering from Seoul National University (SNU), Republic of Korea, in 1977, and the MS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 1979 and 1984, respectively. Since 1984, he has been a faculty member of Computer Science Department at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include micro-architecture, parallel processing, cluster computing, and embedded systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.