

A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation

SANG-WON LEE

Sungkyunkwan University

DONG-JOO PARK

Soongsil University

TAE-SUN CHUNG

Ajou University

DONG-HO LEE

Hanyang University

SANGWON PARK

Hankook University of Foreign Studies

and

HA-JOO SONG

Pukyong National University

Flash memory is being rapidly deployed as data storage for mobile devices such as PDAs, MP3 players, mobile phones, and digital cameras, mainly because of its low electronic power, nonvolatile storage, high performance, physical stability, and portability. One disadvantage of flash memory is that prewritten data cannot be dynamically overwritten. Before overwriting prewritten data, a time-consuming erase operation on the used blocks must precede, which significantly degrades

This work was supported in part by MIC & IITA through IT Leading R&D Support Project, in part by MIC & IITA through Oversea Post-Doctoral Support Program 2005, in part by the Ministry of Information and Communication, Korea under the ITRC support program supervised by the Institute of Information Technology Assessment, IITA-2005-(C1090-0501-0019), and also supported in part by Seoul R&D Program(10660).

Authors' addresses: Sang-Won Lee, School of Information and Communications Engineering, Sungkyunkwan University, Suwon 440-746, Korea; email: swlee@acm.org; Dong-Joo Park, School of Computing, Soongsil University, Seoul 156-743, Korea; email: djpark@ssu.ac.kr; Tae-Sun Chung, College of Information Technology, Ajou University, Suwon 443-749, Korea; email: tschung@ajou.ac.kr; Dong-Ho Lee, Department of Computer Science and Engineering, Hanyang University, Ansan 426-791, Korea; email: dhlee72@cse.hanyang.ac.kr; Sangwon Park, Information Communication Engineering, Hankook University of Foreign Studies, Yongin 449-791, Korea; email: swpark@hufs.ac.kr; Ha-Joo Song, Division of Electronic, Computer, and Telecommunication, Pukyong National University, Busan 608-737, Korea; email: hajuosong@pknu.ac.kr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2007 ACM 1539-9087/2007/07-ART18 \$5.00 DOI 10.1145/1275986.1275990 <http://doi.acm.org/10.1145/1275986.1275990>

the overall write performance of flash memory. In order to solve this “erase-before-write” problem, the flash memory controller can be integrated with a software module, called “flash translation layer (FTL).” Among many FTL schemes available, the log block buffer scheme is considered to be optimum. With this scheme, a small number of log blocks, a kind of write buffer, can improve the performance of write operations by reducing the number of erase operations. However, this scheme can suffer from low space utilization of log blocks. In this paper, we show that there is much room for performance improvement in the log buffer block scheme, and propose an enhanced log block buffer scheme, called FAST (full associative sector translation). Our FAST scheme improves the space utilization of log blocks using fully-associative sector translations for the log block sectors. We also show empirically that our FAST scheme outperforms the pure log block buffer scheme.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Mass Storage; B.4.2 [**Input/Output Devices**]: Channels and Controllers; D.4.2 [**Storage Management**]: Secondary Storage

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Flash memory, FTL, address translation, log blocks, associative mapping

ACM Reference Format:

Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., and Song, H.-J. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embedd. Comput. Syst.* 6, 3, Article 18 (July 2007), 27 pages. DOI = 10.1145/1275986.1275990 <http://doi.acm.org/10.1145/1275986.1275990>

1. INTRODUCTION

Flash memory is being rapidly deployed as data storage for mobile devices, such as PDAs, MP3 players, mobile phones, and digital cameras, mainly because of its small size, low power consumption, shock resistance, and nonvolatile memory [Douglass et al. 1994; Kim et al. 2002; Gal and Toledo 2005]. Compared to a hard disk with the inevitable mechanical delay in accessing data (that is, seek time and rotational latency), flash memory provides fast uniform random access. For example, the read and write time per sector (typically, 512 bytes) for NAND-flash memory is 15 and 200 μ s [Samsung Electronics 2005], respectively, while each operation in contemporary hard disks takes around 10 ms [Hennessy and Patterson 2003].

However, flash memory suffers from the write bandwidth problem. A write operation is slower than a read operation by an order of magnitude. In addition, a write operation may have to be preceded by a costly erase operation because flash memory does not allow overwrites. Unfortunately, write operations are performed in a sector unit, while erase operations are executed in a block unit: usually, one block consists of 32 sectors. An erase operation is very time-consuming, compared to a write operation; usually, a per-block erase time is 2 ms [Samsung Electronics 2005]. These inherent characteristics of flash memory reduce write bandwidth, which is the performance bottleneck in flash-based mobile devices.

To relieve this performance bottleneck, it is very important to reduce the number of erase operations resulting from write operations. For this, flash memory vendors have adopted an intermediate software module, called flash translation layer (FTL), between the host applications (in general, file systems) and flash memory [Estakhri and Iman 1999; Kim and Lee 2002; Kim et al. 2002;

Shinohara 1999]. The key role of FTL is to redirect each write request from the host to an empty area that has been already erased in advance, thus softening the limitation of “*erase-before-write*.” In fact, various FTL algorithms have thus far been proposed [Chung et al. 2006] and each FTL performance varies considerably, depending on the characteristics of the applications. Among them, the log block scheme is well known for excellent performance [Kim et al. 2002]. The key idea of this scheme is to maintain a small number of log blocks in flash memory as temporary storage for overwrites. If a collision (an overwrite) occurs at a sector of flash memory, this scheme forwards the new data to an empty sector in the log blocks, instead of erasing the original data block. Since these log blocks act as cushions against overwrites, the log block scheme can significantly reduce the number of total erase operations.

However, when an overwrite occurs in a data block, the write can be redirected only to one log block, which is dedicated for the data block; it cannot be redirected to other log blocks. For a given overwrite, if there is no log block dedicated to its data block, a log block should be selected as a victim and replaced. Thus, if there are a limited number of log blocks, they might have to be frequently replaced for overwrites. Unfortunately, with the log block scheme, the log blocks being replaced usually have many unused sectors. That is, the space utilization, which can be formally defined as “*the percentage of the written sectors in a log block when it is replaced*,” of each log block is low. In our view, low space utilization degrades the performance of the log block scheme. We need to find out the root cause of low space utilization. For this, we view the role of log blocks from a different perspective. The log blocks in the log block scheme could be viewed as “*a cache for overwrites*,” in which a logical sector to be overwritten can be mapped only to a certain log block—its dedicated log block. From this viewpoint, the address associativity between logical sectors and log blocks is of block level. Thus, we call the log block scheme presented by Kim et al. [2002], the block-associative sector translation (BAST) scheme. We argue that this inflexible block-level associativity in BAST is the primary cause of low space utilization of log blocks.

In this paper, we propose a novel FTL scheme that overcomes the low space utilization of BAST. The basic idea is to make the degree of associativity between logical sectors and log blocks higher, thus achieving better write performance. In our scheme, the sector to be overwritten can be placed in any log block and we, therefore, call our scheme the fully-associative sector translation (FAST) scheme. Hereafter, we denote the FAST scheme and the BAST scheme shortly as FAST and BAST, respectively. As in the computer architecture’s CPU cache realm [Hennessy and Patterson 2003], if we view the log blocks as a kind of cache for write operations and enlarge the associativity, we can reduce the write miss ratio and, therefore, achieve better FTL performance. The main contributions of this paper can be divided into three achievements: (1) to identify the major problem of BAST and its root cause, (2) to provide the motivation of FAST and describe its principal idea and algorithms, and (3) to compare the performance of FAST and BAST over various configurations. An interesting result is that FAST can, in a best case, result in more than 50% reduction both in total elapsed time and in total number of erases.

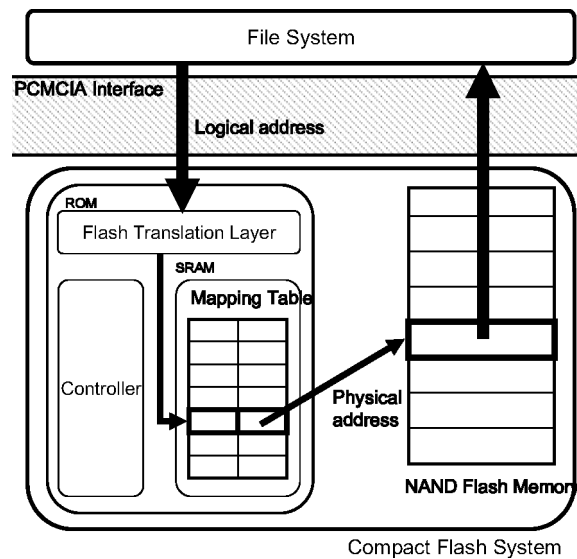


Fig. 1. An anatomy of the NAND flash memory system.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of flash memory and FTL and explains BAST and its disadvantages. Section 3 describes the key idea of FAST and its algorithms. Section 4 compares the performance of our scheme with that of BAST. Finally, Section 5 concludes this paper and outlines future work.

2. BACKGROUND

In this section, we provide a brief overview of flash memory and FTL, and explain in detail the principles behind BAST and its disadvantages. The latter part is somewhat long, but we need to explain BAST in detail because the scheme is not well known in the computer science field.

2.1 Flash Memory and the Flash Translation Layer

Figure 1 shows the general organization of a NAND-type flash memory system. It consists of one or more flash memory chips, a controller mainly executing FTL codes in ROM, an SRAM maintaining address-mapping information, and a PCMCIA host interface. The host system views flash memory as a hard disklike block device and thus issues read or write commands along with “*logical*” sector addresses and data. FTL translates the commands into low-level operations, such as read, write, and erase, using “*physical*” sector addresses. During the address translation, FTL looks up the address-mapping table in SRAM. When issuing overwrites, FTL redirects the physical address to an empty location (free space), thus avoiding erase operations. After finishing an overwrite operation, FTL changes the address-mapping information in SRAM. The outdated block can be erased later.

Besides the address translation from logical sectors to physical sectors, FTL carries out several important functionalities, such as guaranteeing data consistency and uniform wear leveling (or so called block recycling). FTL should be able to maintain a consistent state for data and metadata, even when flash memory encounters an unexpected power outage. For uniform wear leveling, FTL should make every physical block erased as evenly as possible without performance degradation. This functionality of uniform block usage is essential, because there is a physical upper limit on the maximum number of erases allowed for each block (usually, 100,000 times). If a block is erased above the threshold, the block may not function correctly, and thus it is marked as invalid. Even though the block may still work, FTL marks it as invalid for the safety of flash memory. Therefore, if only some physical blocks have been intensively erased, they become unusable very quickly, therefore reducing the durability of flash memory. Therefore, FTL need to use all the blocks as uniformly as possible. Even though these kinds of FTL functionalities are important issues, they are beyond the scope of this paper. In this paper, we focus on the performance issue of the address-mapping technique.

Let us revisit the address-mapping issue. The mapping between the logical address (which the file system uses to interface with flash memory) and the physical address (which the flash controller actually uses to store and retrieve data) can be managed at the sector, block, or hybrid level. In sector-level address mapping, a logical sector can be mapped to any physical sector in flash memory. In this respect, this mapping approach is very flexible. However, its main disadvantage is that the size of mapping table is too large to be viable in the current flash memory-packaging technology. For example, let us consider a flash memory of 1 GB size and a sector size of 512 bytes, which has 2 million sectors. In this case, the mapping information is too large to maintain in SRAM. In block-level address mapping, a logical sector address consists of a logical block number and an offset. The mapping table maintains only the mapping information between logical and physical blocks, while the offsets of the logical and physical blocks are identical. Therefore, the size of block-mapping information is very small, compared to the size of sector-level mapping information. However, this approach also has a serious pitfall: when an overwrite for a logical sector is necessary, the corresponding block is remapped to a free physical block. The overwrite is done at the same offset in the new physical block, the other sectors of the original data block are copied to the new physical block, and ‘finally’ the original physical block should be erased. With regard to block-level mapping, this “*erase-before-write*” problem is an inevitable performance bottleneck, mainly because of the inflexibility in address mapping. In order to overcome the disadvantages of the sector-level and block-level mapping approaches, several hybrid approaches, including BAST, have been proposed. In the hybrid scheme, in addition to a block-level mapping table, a sector-level mapping table for a limited number of blocks is maintained. Thus, it satisfies the size limitations of mapping information and also mitigates the “*erase-before-write*” problem drastically. For a detailed discussion on this issue, please refer to Chung et al. [2006] and Gal and Toledo [2005].

2.2 The BAST Scheme: An Overview

File systems view flash memory as a set of logical sectors—a hard disklike block device. The write function for flash memory can be defined as follows: $write(lsn, data)$, which means “write a given sector $data$ at the logical sector lsn .” When receiving a write request from a file system, FTL finds a physical location in flash memory to write the sector as follows. First, it calculates the logical block number (lbn) using the given lsn .¹ It then retrieves the physical block number (pbn) corresponding to the lbn from the block-level mapping table.² Next, FTL calculates the offset of the retrieved physical block where the sector data will be written.³ Finally, it writes the sector data at the resulting offset in the data block (which is another representation of the physical block).

If the target sector in the data block is already written, FTL writes the given sector data at the same offset in a free-block allocated with the free-block list, and copies all the other written sectors in the data block to the free block. Then, FTL erases the data block and returns it to the free-block list. Whenever a collision between the current write and the previous writes occurs, a large number of sector copies and erase operations are inevitable. This occurrence is called a *merge* operation. To address this problem, many FTL techniques have been suggested. Among them, BAST is known to be the best FTL technique [Chung et al. 2006]. When encountering a collision, BAST reduces the number of merge operations by writing data to temporary storage, called log blocks. In the following, we describe BAST in detail using an example in Figure 2.

In Figure 2, we assume that the number of sectors per block is four and the number of log blocks is two. The upper-left part in Figure 2 indicates a sequence of writes from the file system, and the upper-center and the upper-right part shows the block-level and the sector-level mapping table, respectively. Both mapping tables are usually maintained in SRAM. When the first write operation is called, the BAST algorithm gets data block 10 from the block-level mapping table using logical block 1 ($= 4 \text{ div } 4$). It then stores a given sector at offset 0 ($= 4 \text{ mod } 4$) in data block 10. The second write operation proceeds similarly. In case of the third write operation, BAST finds a collision in data block 10, and, therefore, writes the sector at the first sector in a log block (i.e., $pbn = 20$), dedicated to logical block 1 from the log block list. In case of the fourth write operation, the sector is directed to the next empty sector in the log block. The following write operations will generate the second log block ($pbn = 30$) and the sector-level mapping table.

A collision can occur in other data blocks, for example, data block 12 in Figure 2. In this case, since there is no available log block for data block 12, BAST selects, erases, and returns one of the log blocks (we call it a victim log block). Before returning the victim log block, it needs to merge the victim log block and its data block: BAST copies the up-to-date sectors from the two blocks to a free block, and exchanges the free block with the original data block. In

¹ $lbn = (lsn \text{ div } \# \text{ sectors_per_block})$.

²BAST maintains both block-level and sector-level address mapping tables, which exist in the data block and the log block, respectively.

³ $offset = (lsn \text{ mod } \# \text{ sectors_per_block})$.

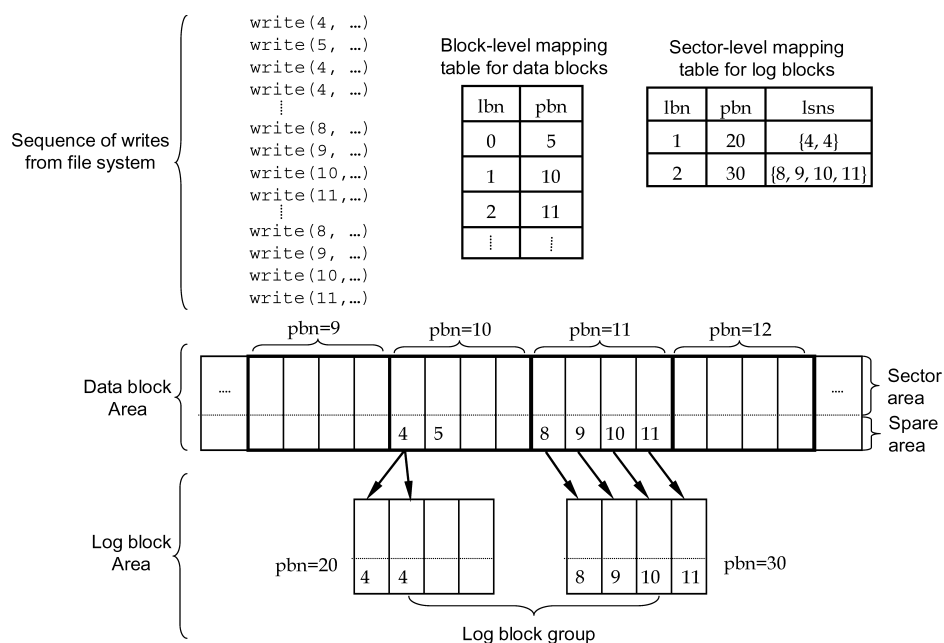


Fig. 2. Processing write operations in the BAST scheme.

addition, BAST updates the block-level mapping table and removes the entry corresponding to the victim log block from the sector-level mapping table. BAST then, erases both the victim log block and its data block, and returns them to the free-block list. If a new log block is necessary, it is allocated from the free-block list. One interesting point is that the merge operation can be optimized if the victim log block satisfies a condition. For example, in Figure 2, all the writes in the log block of $pbn = 30$ are sequential and the total number of written sectors is equal to the capacity of a block. In this case, instead of copying sector data from the victim log block to the free block, we can complete a merge operation just by exchanging the victim log block with its data block; this optimization is called a “switch” operation [Kim et al. 2002].

2.3 Another View of the Log Block Scheme: A Cache for Writes

In this section, we see the role of log blocks from another perspective, and this view will help us to understand the disadvantages of the log block scheme. For this, we would like to briefly explain the principal role of the CPU cache and its basic functions. Because of the principle of locality in data access of computer programs [Hennessy and Patterson 2003], a small cache combined with the memory hierarchy can provide users with a very fast, large, and cheap memory system (Figure 3a). In this respect, the log blocks in the log block scheme can also be viewed as “a cache for overwrites” (Figure 3b). In case of collisions, we can complete the write operation much faster by writing sectors in the log blocks (that is, the cache), instead of overwriting its original data block.

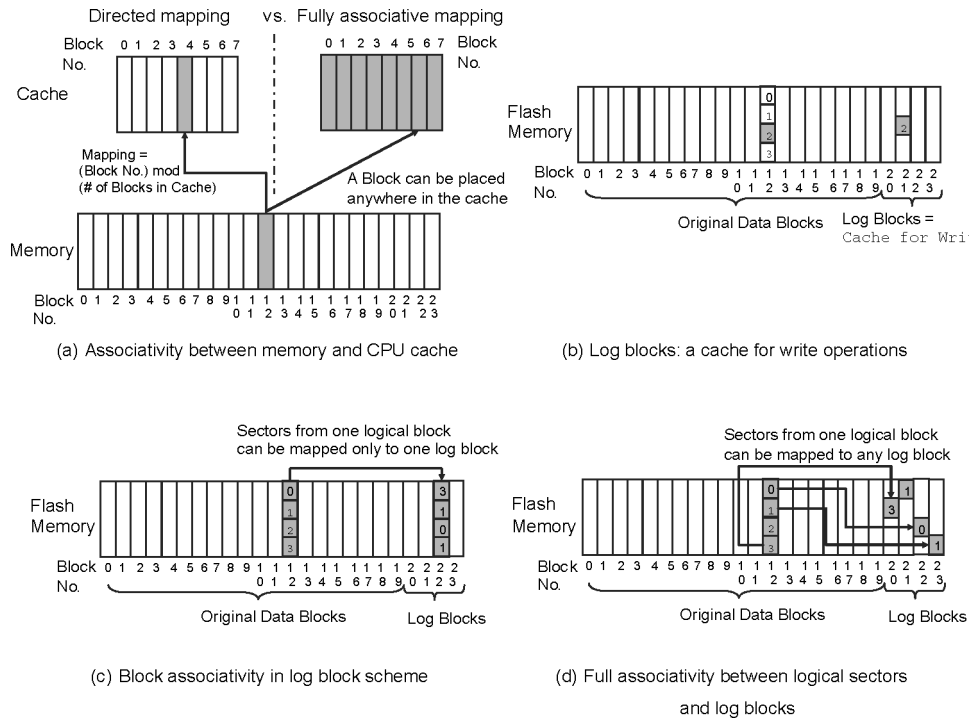


Fig. 3. CPU cache versus log blocks: associativity.

As depicted in Figure 3a, there is the associativity issue between memory and cache regarding how memory blocks are mapped to cache blocks. There are various associativity schemes used in computer architecture, including direct-mapped, n-way associative, and fully associative approaches [Hennessy and Patterson 2003]. With the direct-mapped approach, a memory block can be placed only in a dedicated cache block via a mathematical formula, while the fully associative approach allows a block to be placed anywhere in the cache. In terms of the associativity, we could say that the log block scheme takes the block-associative sector translation approach, because the scheme directs all the overwrites for one logical block only to its dedicated log block, as depicted in Figure 3c.

2.4 Disadvantages

This block-level associativity of BAST results in two performance problems. The first problem is analogous to the high miss ratio in direct-mapped associativity between CPU cache and main memory, which results in the block-thrashing [Hennessy and Patterson 2003]. BAST has a similar block-thrashing problem. If the cache in flash memory (namely, the log blocks) cannot accommodate all collisions during the execution of a host application (especially, writes for hot blocks in flash memory), BAST will experience numerous capacity misses, thus causing the block thrashing. For instance, assume that two blocks (and four sectors per each block) are allocated for the cache and that an overwrite pattern

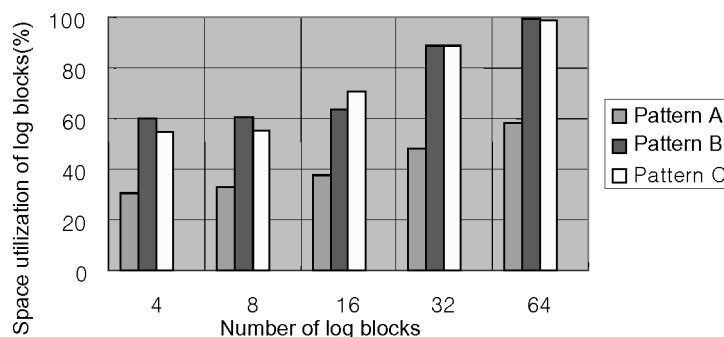


Fig. 4. Low space utilization in the BAST scheme.

is the sequence “S0, S4, S8, S12, S0, S4, S8, S12,” where S_i is the logical sector number for each overwrite and the original data of each logical sector is already stored in the data block. In addition, we assume that all the original data blocks are already filled. Please note that S0, S4, S8, and S12 come from different blocks, because each block holds four sectors. For every write, starting from the first S8 overwrite, BAST should replace either of the two log blocks. We refer to this phenomenon as *log block thrashing*. Each victim replacement will accompany an expensive merge operation. Moreover, every victim log block in this example holds only one sector of data when it is replaced; the other three sectors remain empty.

Another performance problem arises from block-level associativity. Assume that under the same cache above, the overwrite operations for one hot logical block occur successively, e.g., the write pattern is the sequence “S0, S2, S1, S3, S1, S0, S2, S3.”⁴ In BAST, for every fourth write, the log block allocated to the hot logical block should be merged with its original data block, although the other log block is idle. In summary, when intensive overwrites for one hot block occur during a given time window, the log block scheme might result in increased write operations.

Because of these two problems, the log blocks in BAST would show very low space utilization when they are replaced from the log buffer. Each sector in the log block is a very precious resource in the sense that it might prevent an “erase-before-write” phenomenon if exploited properly. Low space utilization of the log blocks indicates that there is an opportunity for performance improvement in BAST. In order to measure the severity of low space utilization, we simulate BAST over various workloads and calculate the space utilization of the log blocks being replaced. We used three workloads: pattern A of mainly random writes generated from a Symbian environment and pattern B and C, mainly of sequential writes generated from a digital camera. For the experiment, we measured the space utilization by increasing the number of log blocks from 4 to 64. Figure 4 shows the experimental result for space utilization in BAST. As shown in the figure, the space utilization for pattern A is under 50%, even when

⁴The hot logical block number is 0.

the log block number is sixteen. The situation is not much better with patterns B and C.

As defined in Section 1, the space utilization is “the percentage of the written sectors in a log block when it is being replaced.” With this definition in mind, let us analyze the result in Figure 4. When the number of log blocks is less than the number of hot data blocks (that is, the data blocks of which some sectors are actively overwritten) at a certain point of time, each log block might become a victim so quickly that it has little chance to fill itself with corresponding sectors. However, as more log blocks are available, each log block could stay longer in the log buffers, and thus has more chance to fill itself with more sectors. This is the reason why the average space utilization of the log blocks increases with the number of log blocks.

For a fixed number of log blocks, the space utilization in BAST gets worse as the write pattern becomes more random (e.g., in Figure 4, pattern A is more random than patterns B and C). This can be explained as follows: because a more random pattern implies more hot blocks at a certain point of time and thus experiences more capacity misses, each log block has less chance to fill itself. Our concern is whether we can improve the utilization of the precious free sectors in log blocks, thus achieving high performance, even with a limited number of log blocks (less than 10).

3. FAST: A LOG BUFFER SCHEME USING FULLY-ASSOCIATIVE SECTOR TRANSLATION

3.1 Key Idea

Based on the discussion in Section 2, we can consider a natural extension of the log block scheme: “*What if we take the fully associative approach in mapping logical sectors to log blocks?*” In this approach, a logical sector can be placed in any log block, which gives two performance optimization opportunities. The first one is to alleviate log block thrashing. Even if the number of hot logical blocks in a given time window is greater than that of log blocks, a higher degree of associativity between logical sectors and log blocks can help reduce the miss ratio in finding an empty sector in the log blocks. This is very similar to the reduction in the cache miss ratio when the associativity between memory blocks and cache blocks increases [Hennessy and Patterson 2003]. For example, under the fully-associative approach, the write sequence “S0, S4, S8, S12, S0, S4, S8, S12” in Section 2.4 does not require any log block replacement and thus does not require any erase or merge operation.

The second (and more important) optimization is that we can avoid many merge operations, which are inevitable, in BAST when a dedicated log block has no sector to accommodate overwrites. Consider again the write sequence “S0, S2, S1, S3, S1, S0, S2, S3” in Section 2.4 and assume that the logical sectors S0, S1, S2, and S3 are already written to the corresponding data block. If we adopt the fully-associative mapping approach, two log blocks are sufficient to cover all the write operations. We can avoid a costly merge operation for every $(4th + 1)$ write (which is inevitable in BAST) and can also delay the merge

operations until there is no empty sector in the log blocks. In summary, when one of the logical blocks is very hot in a given time period, the full associativity approach can avoid many merge operations.

Even though these optimizations might seem to be naïve, the performance impact is considerable. In subsequent sections, the further advantages of full associativity between logical sectors and log blocks will be described in detail.

3.2 The Architecture

The architecture of FAST is analogous to that of BAST. One important difference is that log blocks in FAST are divided into two areas: one log block is used for sequential writes and the other blocks for random writes. Let us explain the reason why a log block is dedicated for sequential writes. As stated in Section 2.2, a switch operation is more efficient than a merge operation. Since most of the workload traced from flash memory systems contains a large number of sequential writes, it is likely that many switch operations arise from them. However, since our FAST scheme takes the fully-associative address-mapping approach, we cannot take advantage of switch operations unless sequential writes are handled separately from random writes. That is, if both types of write patterns are intermixed only in a fully associative manner, there is little chance for switch optimization in FAST. For this reason, FAST keeps a dedicated log block for sequential writes. In the subsequent sections, the SW log block denotes the log block for sequential writes and the RW log blocks represent the log blocks for random writes.

The other architectural difference between FAST and BAST is that FAST maintains separate sector-level mapping tables for the above two types of log blocks. The sector-mapping table for the SW log block keeps information on which logical block the current log block corresponds to, and how many sectors are currently stored in the log block. Meanwhile, a more complex mapping table is needed for the RW log blocks. This table records which sectors are written and the offsets of the RW log blocks. In theory, a sector can be placed on any RW log block. However, we take a rather simple approach of filling the RW log blocks with sectors in sequence.

This architecture of FAST yields a small overhead in sector read operations, compared to BAST. For every sector read operation in FAST, we should check whether the most recent version of the sector exists in the log blocks. You need to note that this check is accomplished by scanning the sector-level mapping table in SRAM, not by scanning the log blocks. Thus, for every read operation against sectors in log blocks, FAST yields the overhead for scanning the sector-level mapping table in SRAM. If we assume 128 MB flash memory with 512 byte-sized sectors, there are 256 K sectors. Hence, we need 18 bits to represent an lsn (logical sector number), and each entry in sector-level mapping table can be stored in 4 bytes. If we assume 6 log blocks and each block has 32 sectors, then the total number of entries in sector-level mapping table is 192. If we scan the sector mapping table from the end of the mapping table in order to check whether a logical sector is in the table, we need to read 96 entries from

SRAM, on average. The contemporary 4-byte read time from SRAM is 10 ns⁵ [Hennessy and Patterson 2003], and thus the average time of a scan becomes 960 ns (about 1 μ s), and this overhead is small compared to the read time for a sector, which is typically 15 μ s [Samsung Electronics 2005]. BAST also has the overhead of memory scan for sector read operations because they also maintain a sector-level mapping table, as shown in Figure 2, but they need to scan only the sector-mapping information of a log block, and not the entire sector-level mapping table.

Finally, consider the issues of meta-data and data consistency. FAST is exactly the same to BAST in managing the block-level mapping table. That is, block-level mapping information is stored in some dedicated blocks of flash memory, called “*map blocks*,” and the block-level mapping table is cached in SRAM for the fast lookup of mapping information at runtime [Kim et al. 2002]. The consistency between block-mapping tables in SRAM and in flash memory can be achieved as in BAST. Besides the block-mapping table, FAST maintains the sector-mapping table in SRAM. We need to consider how to achieve the consistency of a sector-mapping table when we encounter an unexpected power-off. In the earlier FTL schemes, such as M-system [Ban 1995], with sector-level address mapping, they record the mapping information between a logical sector and a physical sector in the spare area of the physical sector. From the sector-mapping information in the spare area, the sector mapping table in SRAM is dynamically constructed during the booting time. Our sector-level mapping table also can be maintained in this way.

3.3 Handling Write Operations

In this subsection, we explain how FAST handles the write operations issued from the file system. As mentioned in Section 3.2, the log blocks in FAST are divided into two groups: the SW and the RW log blocks. When a collision occurs in a specific data block, the corresponding sector is directed to either of two groups, as depicted in Figure 5. For every sector to be written, FAST first examines whether its insertion into the SW log block will result in a switch operation. If so, the sector data is appended in the SW log block; otherwise, the data is written in the RW log block.

If a log block is to be a candidate of switch operation, it must satisfy two conditions: (1) the lsn at the first offset in the log block is divided by the number of sectors per block (for example, 4), that is, $lsn \bmod 4 = 0$, and (2) the log block is filled up with the sectors, which are sequentially written from the beginning offset to the final offset. Thus, FAST directs only the sectors satisfying either condition into the SW log block.

When a collision occurs in a data block, the corresponding sector is among one of three cases in Figure 5, where we assume the number of sectors per

⁵This value of access time is in case of random access time. Therefore, the average access time for each of 96 entries, in case of sequential scan, would be much less. In addition, we believe that several hundred bytes of all the sector-mapping table can be cached in L1 cache of the CPU used in the FTL controller (e.g., ARM), and the real overhead will be much smaller than the estimated overhead in this paper.

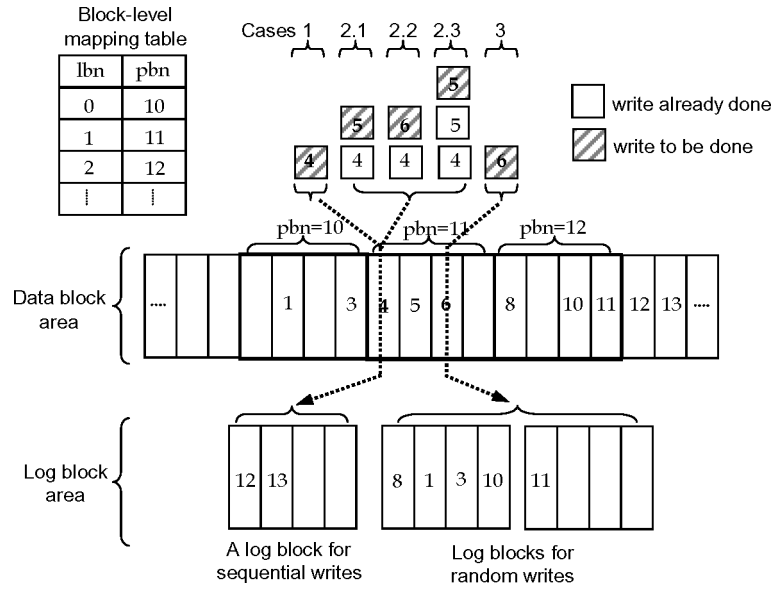


Fig. 5. Performing write operations in the FAST scheme.

block is four and two log blocks exist for random writes. First, if the sector to be overwritten satisfies the first condition, then it is inserted into the SW log block. If the SW log block is already occupied by other sectors, it is merged with its data block, erased, and then returned to the free-block list. A new block is allocated for the SW log block and the sector is inserted into the new block. In case 1 of Figure 5, sector 4 satisfies the first condition and is inserted into the SW log block. However, since other sectors (sectors 12 and 13) exist in the log block, merge and erase operations for the log block must precede the insertion.

If the sector to be overwritten satisfies the second condition, it is sequentially inserted into the next empty location of the SW log block. It is exemplified in case 2.1 of Figure 5, where we assume that the sector 4 is already written to the log block prior to the sector 5. Since the sector 5 satisfies the second condition, it is inserted into the SW log block. On the other hand, sectors 6 and 5 of the cases 2.2 and 2.3 do not satisfy the second condition. If these sectors are inserted into the SW log block, we cannot apply the switch optimization to the log block and, therefore, simply merge the SW log block with its data block. Only if the SW log block is filled with the sectors satisfying one of the two conditions, can FAST perform a switch operation.

Finally, if the sector to be overwritten does not satisfy either condition, then it is inserted into one of the RW log blocks. As previously described, the sector data is appended to the next empty location in the RW log blocks. If there is no empty sector in the RW log blocks, FAST selects a victim block in a round-robin fashion, and merges it with corresponding data blocks. We would like to remind the readers that there could be more than one corresponding data block for a victim block because of the fully-associative mapping in FAST. In turn, FAST erases the victim log block and returns it to the free-block list. In case 3 of

Figure 5, since the sector 6 does not satisfy either condition, it is inserted into the RW log blocks.

Figure 6 describes an algorithm of the write operation issued from the file system to FTL. In Figure 6, Algorithm 1 shows how the function `write()` works, which, in turn, calls the function `writeToLogblock()` in Algorithm 2 when a collision occurs. If there is no collision, the sector data is simply written to its data block. Algorithm 2 works as follows. If the given offset is zero, according to another condition, FAST executes a switch operation or a merge operation and then stores the given data into the new SW log block, which corresponds to the case 1 of Figure 5. If the offset is not zero, the given sector data is appended to the SW log block (line 13 in Algorithm 2), written to the new block by the merge operation (line 15 in the Algorithm 2), or put into the RW log block (line 20 in the algorithm 2). These correspond to cases 2.1, 2.2, 2.3, and 3 in Figure 5, respectively. In the next subsection, we describe in detail the merge operation for the log blocks and the victim selection issue for RW log blocks.

3.4 Handling Merge Operations

All three cases mentioned in the previous subsection give rise to a merge or switch operation. As explained in Section 2.1, a merge operation is achieved in three steps: (1) to copy the up-to-date sectors from the log block and its corresponding data block to a free-block, (2) to change the mapping information, and (3) to erase both the data block and the log block and return them to the free-block list. In order to optimize a merge operation, it is necessary to minimize the number of copy and erase operations. This section describes how FAST handles switch operations and merge operations.

- **A switch operation in the SW log block:** This case is similar to the switch operation in BAST. If the SW log block is filled up with the sequential sectors, FAST exchanges it with its data block, erases the data block, and returns the erased data block to the free-block list.
- **A merge operation in the SW log block:** In FAST, there are two cases which require a merge operation in the SW log block: (1) when the SW log block encounters a sector satisfying the first condition in Section 3.3, and (2) when the SW log block encounters a sector violating the second condition of Section 3.3. The merge operation of FAST is analogous to that of BAST, but we can apply another optimization technique to the merge operations in FAST when the SW log block has a particular state. For instance, in Figure 7a, consider the SW log block containing sectors of S4, -1, S6, -1 where -1 indicates “no sector written.” In this case, FAST will copy the data of empty sectors with -1’s (sector 5, in this example) from its original data block, exchange the updated log block with its data block, and, finally, erase the data block and return the erased data block to the free-block list. Please note that this optimization is distinct from the switch operation in BAST. This optimization requires a small number of copies and only one erase operation, while the original merge operation requires a sectors-per-block number of copies and two erase operations for the SW log block and its original data

```

Algorithm 1 write(lsn, data)      /* data are logically written to the sector of lsn */
1  lbn := lsn div SectorsPerBlock;
2  offset := lsn mod SectorsPerBlock;
3  pbn := getPbnFromBMT(lbn);      /* get pbn from block-level mapping table */
4  if a collision occurs at offset of the data block of pbn
5      call writeToLogblock(lsn, lbn, offset, data);
6  else
7      write data at offset in the data block of pbn;
8  end if

Algorithm 2 writeToLogblock(lsn, lbn, offset, data)
1  if offset is zero                /* Case 1 in Figure 5 */
2      if there are no empty sectors in the SW log block
          /* the log block is filled with sequentially written sectors */
3      perform a switch operation between the SW log block and
          its corresponding data block;
          /* after switch, the data block is erased and returned to the free-block list */
4      else
          /* before merge, a new block is allocated from the free-block list */
5      merge the SW log block with its corresponding data block;
          /* after merge, the two blocks are erased and returned to the free-block list */
6      end if
7      get a block from the free-block list and use it as a SW log block;
8      append data to the SW log block;
9      update the SW log block part of the sector-mapping table;
10 else
11 if the current owner of the SW log block is the same with lbn
12     last_lsn := getLastlsnFromSMT(lbn);      /* SMT: sector-mapping table */
13     if lsn is equivalent with (last_lsn+1)  /* Case 2.1 in Figure 5 */
14         append data to the SW log block;
15     else                                     /* Case 2.2 and 2.3 in Figure 5 */
16         merge the SW log block with its corresponding data block;
17         get a block from the free-block list and use it as a SW log block;
18     end if
19     update the SW log block part of the sector-mapping table;
20 else                                         /* Case 3 in Figure 5 */
21     if there are no rooms in the RW log blocks to write data
22         select the first block of the RW log block list as a victim;
23         merge the victim with its corresponding data block;
24         get a block from the free-block list and add it to the end of
          the RW log block list;
25         update the RW log block part of the sector-mapping table;
26     end if
27     append data to the RW log blocks;
28 end if
29 end if

```

Fig. 6. Write algorithm in FAST.

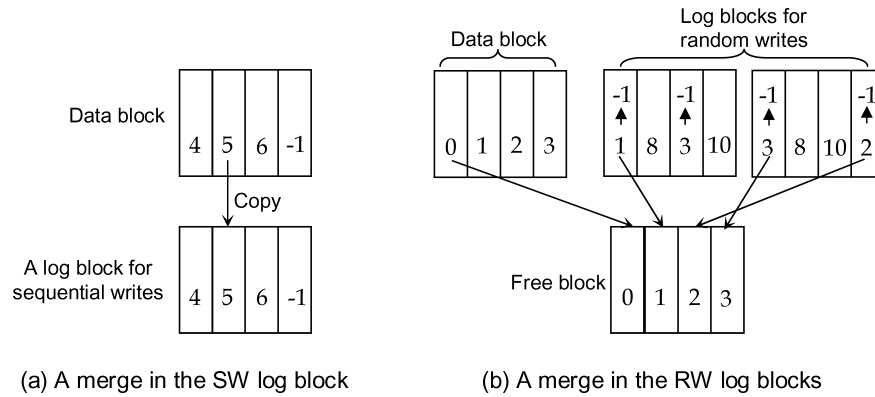


Fig. 7. Performing merge operations in FAST.

block. This optimization could also be applied to BAST, but we first suggest the technique in this paper.

- A merge operation in the RW log blocks:** If no more empty sectors exist in the RW log blocks, FAST chooses one of the RW log blocks as victim and merges the victim block with its corresponding data blocks. The victim selection is done in a round-robin fashion. With regard to merge operations in the RW log blocks, the readers should note that the sectors in a victim might originate from several different logical blocks and, therefore, the number of merge operations per a victim block is equal to the number of logical blocks corresponding to the sectors in the victim.⁶ In FAST, each merge operation per logical block proceeds as follows. First, in order to find all sectors for the logical block, FAST scans the sector-level mapping table for the RW log blocks. For each sector found, FAST copies the most up-to-date version from the RW log blocks to a free-block. Then, FAST marks all the found sectors in the sector-level mapping table as invalid state (-1). If a sector with invalid state is encountered in the future victim block, we can ignore the sector, because we know that its up-to-date version is already stored in its data block. Next, FAST selectively fills each empty sector in the free-block with its corresponding sector in the data block, and, in turn, exchanges the free-block with the data block. The data block is erased and returned to the free-block list. Finally, after merging all the logical blocks in the victim, FAST erases the victim log block itself and returns it to the free-block list. For example, let us consider Figure 7b, where the first log block is a victim. Since the sectors in the victim block come from two different logical blocks 0 (for S1 and S3) and 2 (for S8 and S10), two merge operations for (S1, S3) and (S8, S10) are required. The merge operation for (S1, S3) proceeds as follows: The up-to-date sectors of logical block 0, i.e., sector 1 in the victim block and sectors 2 and 3 in the nonvictim log blocks, are copied to a free block. Then, the sector 0 in the original data block is copied to the free block. The data block is then

⁶In contrast, all the sectors in each victim log block in BAST originates from one logical block and, thus, only one merge operation is required per victim.

erased and returned to the free-block list. The merge operation for (S8, S10) proceeds similarly; then, the victim log block is erased and returned to the free-block list.

3.5 An Analytical Comparison to BAST

Before proceeding, we would like to compare BAST and FAST in an analytical way so that the readers can understand the advantages of FAST more clearly. For this, we will analyze why and how the full associativity in FAST can reduce the number of erase operations, compared to BAST, in various cases, in terms of the number of hot data blocks and the number of log blocks. In this subsection, for the brevity of description, we denote the number of hot data blocks as n_{hb} , the number of the log blocks as n_{lb} , and the number of sectors-per-block as n_{spb} . We cover the following four cases: (1) when n_{hb} is equal to n_{lb} , (2) when n_{hb} is one, (3) when n_{hb} is greater than n_{lb} , and (4) when n_{hb} is between one and n_{lb} . In this subsection, we will not take large sequential write patterns into considerations, because both BAST and FAST show a similar performance characteristic for the patterns and, instead, focus on the random write patterns. In case of FAST, the n_{lb} merely indicates the number of the RW log blocks, except for the SW log block, because we do not consider the sequential write patterns.

- **When n_{hb} is equal to n_{lb} .** In this case, BAST and FAST will require nearly the same number of erase operations. From the perspective of BAST, each sector to be overwritten has its dedicated log block and requires a merge operation only when its dedicated log block is full. Therefore, a merge operation is necessary, on average, in every n_{spb} th sector writes, and a merge will result in two erase operations: one for the data block and the other for the log block. FAST also requires a merge operation in every n_{spb} th sector writes, that is, when a new log block is full. In order to merge the first victim, FAST needs the $(n_{lb} + 1)$ erase operations if we assume that the random write patterns are uniformly distributed: one for the victim log block and the others for all the hot data blocks. However, for each of the following $(n_{lb} - 1)$ victim log blocks, FAST can complete the merge operations of the victim by erasing only the victim block, since almost of the sectors were already marked as invalid during the merge operation of the first victim. Thus, FAST will require approximately $2 * n_{lb}$ erase operations by when the first n_{lb} log blocks are merged and, thus, on average, each victim replacement requires two erase operations, as in BAST.
- **When there exists only one hot data block.** In an extreme case where all the sectors to be overwritten come from only one logical block (that is, one hot data block), BAST writes all the sectors only in a dedicated log block and thus requires a merge operations in every n_{spb} th write, even though other log blocks are empty. In contrast, FAST utilizes every log block and thus accepts the sector writes until it all the log blocks are filled. In addition, while merging the first victim block, FAST invalidates all the sectors in other log blocks and, thus, it can complete the merge operations of the following victim blocks by erasing only the victim block, because all the sectors in the

following victims were already marked as invalid during the merge operation of the first victim. Thus, by the time the first n_{lb} log blocks are merged, FAST can save the $(n_{lb} - 1)$ erase operations, compared to BAST. From this, we could argue that the performance of FAST is scalable to the n_{lb} , because it can save more erase operations as the n_{lb} increases. In contrast, BAST does not benefit from the increased n_{lb} because of its block-level associativity.

- **When n_{hb} is greater than n_{lb} .** In the case where the number of hot blocks at a certain point of time is greater than n_{lb} , BAST should replace a log block whenever a sector without its dedicated log block in the log buffer arrives, and only a small fraction of the victim block is, in most cases, used. That is, the block-level associativity results in log block thrashing and low space utilization of the log blocks. In the worst case, each sector to be overwritten may require a victim replacement, which, in turn, requires two erase operations. However, FAST caches the sectors to be overwritten in a new log block until it is full. When replacing the first victim block, FAST might require the $(n_{spb} + 1)$ erase operations, at most: one for the victim block and the others for the n_{spb} different data blocks. In a case where the n_{hb} is less than the n_{spb} , the number of erase operation will be n_{hb} plus one. As in case of only one hot data block, while merging the first victim block, FAST will invalidate most of the sectors in other log blocks and, thus, complete the merge operations of the following $(n_{lb} - 1)$ log blocks just by erasing the victim block because most of the sectors in the following victim might be already marked as invalid. Thus, FAST requires approximately the $(n_{spb} + n_{lb})$ erase operations by the time the first n_{lb} blocks are replaced. Please note that at the point of time when the first n_{lb} blocks are replaced, exactly $n_{lb} * n_{spb}$ sectors have been overwritten. Meanwhile, for this number of sectors to be overwritten, BAST will, in its worst case, require $(2 * n_{lb} * n_{spb})$ erase operations. With regard to the scalability issue, the performance of BAST does not improve with the n_{lb} until it equals to the number of hot blocks, while FAST can reduce the number of erase operations, because more sectors will be marked as invalid and thus there is more chance to skip erase operations for data blocks.
- **When n_{hb} is between one and n_{lb} .** As the n_{hb} increases from one to n_{lb} , the performance of BAST is approaching that of FAST, because BAST distributes the sectors to be written uniformly over more log blocks and thus the number of erase operations decreases. Meanwhile, the performance of FAST does not improve with the number of hot blocks. When the n_{lb} becomes above the n_{lb} , the performance of BAST becomes radically worse, because of block thrashing.

In summary, compared to BAST, the full associativity between logical sectors and log blocks in FAST is very helpful in reducing the number of erase operations, in all cases, except when n_{hb} is equal to n_{lb} .

Before closing this subsection, we would like to comment on the overhead of merge operations in FAST. Some readers might think that the logic of the merge operations in FAST seems to be quite complex, compared to BAST, and it might have a negative effect on performance. In fact, more than one merge

operations can occur when a victim log block is replaced in FAST, because of its full associativity. For example, let us assume that the log blocks are filled with only two sectors, S_i and S_j , which come from different data blocks. When replacing a victim, FAST executes two merge operations for the two data blocks. (Of course, O-FAST can even skip these merge operations.) However, when the other log blocks are later replaced, we do not need any merge operation because all the sector data are already invalidated during the previous merge operations. That is, even though more than one merge operation are necessary when a victim log block is replaced in FAST, the additional merge operations are executed in advance. Moreover, FAST can skip many merge operations, which are inevitable in BAST. At this point, you might still suspect that a merge operation for a data block requires the full scan of the log blocks, which is another overhead. Instead of scanning the log blocks, however, FAST looks up the sectors of the data block from the sector-mapping table in SRAM and reads only the most recent sector data from the log blocks. Except for a sector-mapping table scan, the sector read time for one merge operation in FAST is exactly the same as in BAST. As described in Section 3.2, it takes about $1 \mu s$ to scan the sector-mapping table, which is negligible, compared to the expensive erase operations in a merge operation. In summary, the seemingly complex logic of merge operations in FAST does not result in real performance overhead.

3.6 O-FAST

The performance of the log block based schemes (both BAST and FAST) is mainly determined by the frequency of merge operations resulting from log block replacement. FAST outperforms BAST because it delays the merge operations until all log blocks are filled and skips some merge operations, reducing the number of merge operations. This recognition brings us to ask whether we can delay the merge operations a little longer or even skip more merge operations. In this subsection, we propose another optimization, which delays the merge operations longer than FAST, called O-FAST (the abbreviation stands for Optimized FAST). The idea of O-FAST can be easily illustrated using an example. Let us consider Figure 7b, in which the victim block has two groups of sectors (S_1, S_3) and (S_8, S_{10}) from two different logical data blocks and thus two merge operations are required in FAST. However, if up-to-date versions of sectors (S_8, S_{10}) exist also in the nonvictim log blocks (i.e, the second log block), we can safely skip the merge operation for (S_8, S_{10}), because the sector data in current victim log block is out of date and we can merge the corresponding data block with more up-to-date sector data. In order to guarantee the consistency between the read and write requests from the file system, FTL has to maintain the up-to-date sectors either in the data blocks or in the log blocks.

Based on this observation, O-FAST can delay (in effect, skip) the merge operation for a logical data block if more recent versions for all those sectors exist in the nonvictim log blocks. In other words, O-FAST postpones the merge operation for a logical data block until we can no longer delay the merge operation. In this respect, we call the merge operation in O-FAST as “*lazy merge*.” In contrast, we call the merge operation in FAST as “*eager merge*,” because FAST initiates

a merge operation for a logical data block whenever it finds any sector for the data block in a victim log block. In Figure 7b, for example, O-FAST delays (that is, skips) the merge operation for sectors (S8, S10) while FAST executes the merge operation immediately. In particular, when the sectors from one logical block are repeatedly and intensively accessed, O-FAST can delay most merge operations for the block, and thus outperforms FAST.

3.7 Performance Analyses Using Examples

In this section, we compare BAST versus FAST and FAST versus O-FAST using the examples in Figures 8 and 9. In the figures, we assume that the number of log blocks is four and the number of sectors per a block is four. In FAST and O-FAST, one log block is used for sequential writes and the other three log blocks for random writes. Regarding the victim selection for log block replacement, BAST and (O-)FAST have a little variation in their strategies. In case of BAST, since the victim selection strategy is not clearly mentioned by Kim et al. [2002], we assume the following: (1) if an overwritten sector has its corresponding log block and the block is full, the log block is selected as victim, (2) if the new sector does not have its corresponding log block, we select a log block to which we can apply the switch optimization as victim, if any, and (3) in other case, we select a victim log block in a round-robin fashion. (O-)FAST selects a victim block from the RW log blocks in a round-robin way. The write pattern is intended to include both small random writes (S1, S9, Figure 8) and large sequential writes (S27, S28, S29, . . . , Figure 8).

In Figure 8, FAST requires seven erase operations to process all the write requests, while BAST needs nine erase operations. Both FAST and BAST require four merge operations. In BAST, each *normal* merge operation requires two erase operations. In FAST, however, even though the first victim replacement requires three erase operations (one for the victim block and two for the data blocks 1 and 3), two merge operations after the write request ⁽¹⁵⁾ result in *simple* merge operations, which requires just one erase operation.

Figure 9 shows how many erase operations O-FAST requires for the same write pattern used in Figure 8. Prior to the write request ⁽¹⁰⁾, O-FAST proceeds like FAST in Figure 8. The first merge operation happens at the write request ⁽¹⁰⁾, where the victim log block is the first block in the RW log blocks. Fortunately, since up-to-date versions of all the sectors in the victim (i.e., S1's and S9's) exist in the nonvictim log blocks, we can delay the merge operations for the two corresponding blocks and just need to erase only the victim block. The other merge operations in the figure require only one erase operation. For this reason, O-FAST incurs five erase operations to process all the write requests, while FAST requires seven erase operations. In summary, the “lazy merge” in O-FAST allows us to skip two erase operations, compared to the “eager merge” in FAST. This is the intrinsic difference between the two schemes.

4. PERFORMANCE EVALUATION

To evaluate and compare the performance characteristics of BAST, FAST, and O-FAST, we developed a simulator for each scheme and performed

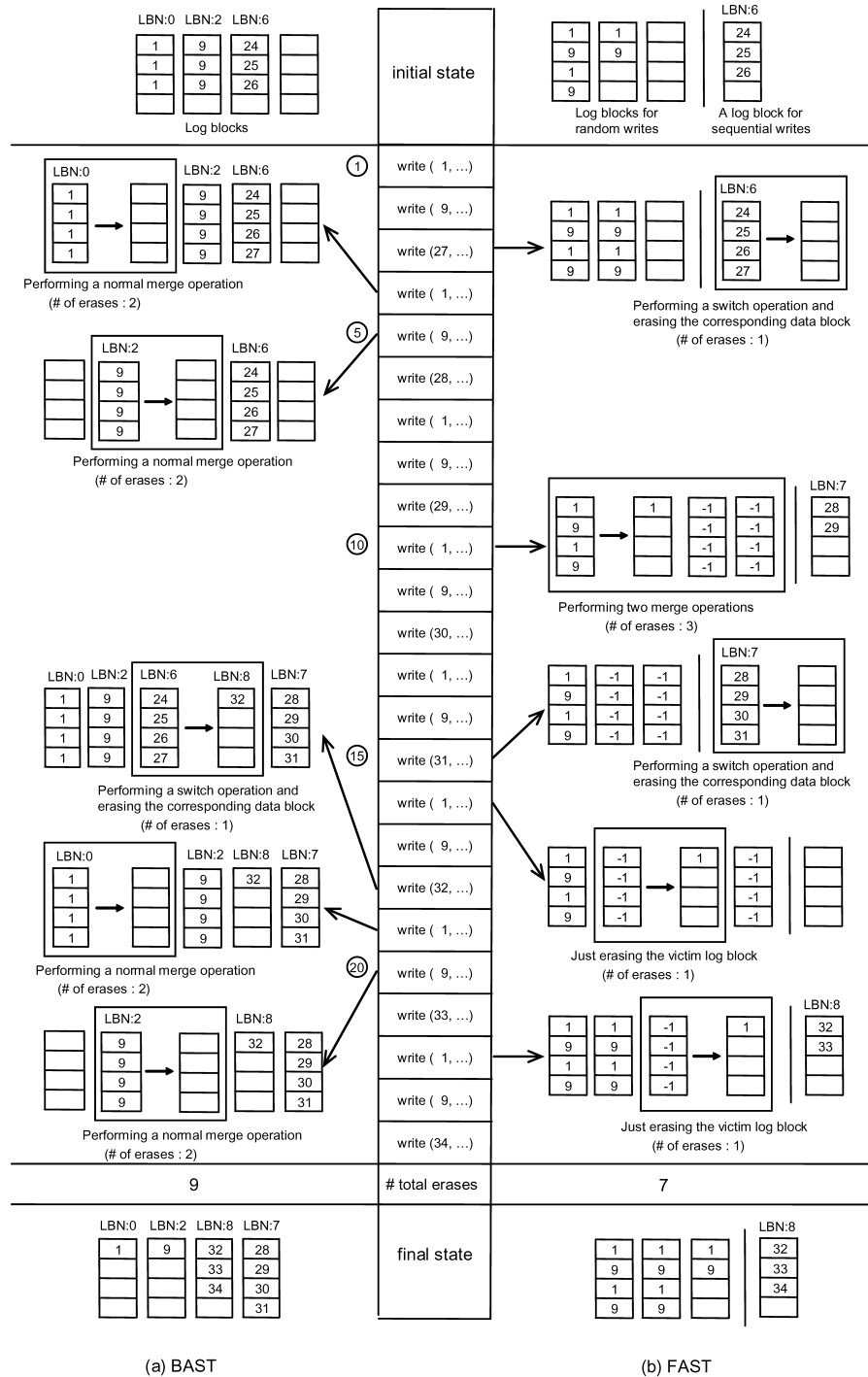


Fig. 8. Performance analysis: BAST versus FAST.

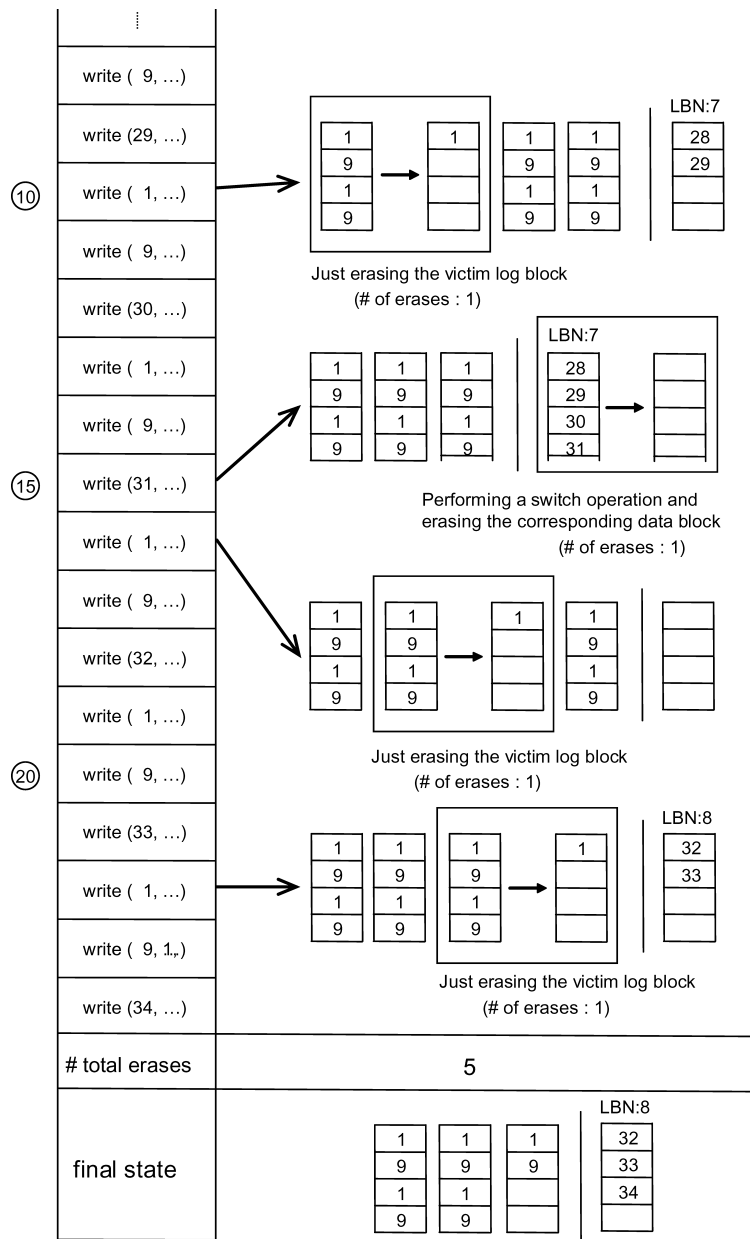


Fig. 9. Performance analysis: O-FAST.

trace-driven simulations. For each given trace, the simulator counts the number of read, write, and erase operations FTL generates, and calculates the total elapsed time using the formula $total_elapsed_time = read_count * read_time + write_count * write_time + erase_count * erase_time$, where we assume $15 \mu s$ for a sector read, $200 \mu s$ for a sector write, and $2 ms$ for a block erase.

Table I. The Traces Used in Experiments

Pattern	Description	# of writes
A	Workload from digital camera (A company)	2,199,200
B	Workload from digital camera (B company)	3,144,800
C	Workload from Linux O/S	398,000
D	Workload from Symbian O/S	404,900
E	Uniform random writes (generated synthetically)	150,000

Table I shows the five traces we used in the experiment: the first four traces (pattern A–D) have been obtained from the authors of BAST [Kim et al. 2002] and the fifth trace (pattern E) contains uniform random writes, which have been synthetically generated. We believe that these patterns are complex enough to show the characteristics of the FTL schemes and to compare them. The random pattern E, in fact, is not a typical workload of contemporary flash memory applications. Nevertheless, because the flash memory is expected to be used as the storage media for more general computer systems with more random write patterns, including laptop computers [Lawton 2006; Paulson 2005], it is meaningful to compare three FTL schemes over a random write pattern.

Figure 10 shows the experimental results of BAST, FAST, and O-FAST using the trace workload, where the total elapsed time is measured in the units of a second. If necessary, we will explain the performance difference of FAST and O-FAST in detail. The performance metrics we used are the number of total erase count and the total elapsed time. Compared to read/write operations, the erase operation is more time consuming and, therefore, the efficiency of an FTL scheme mainly depends on how many erase operations it can avoid. In the experiment, we test the impact of the number of log blocks by increasing the number of log blocks from 4 to 64 for each configuration. FAST beats BAST consistently in all the patterns. Now, we will investigate each case in detail.

First, let us explain the case of pattern E with uniform random writes, in which the advantage of FAST/O-FAST is clearly demonstrated. Each write operation in BAST may result in a costly merge operation, since the write operation can place only in its dedicated log block. Please note that the performance in BAST does not improve with the number of log blocks: because the write pattern is very random (that is, the number of hot blocks is generally more than 64), the space utilization of the log blocks in BAST does not improve. In contrast, each write operation in FAST is appended in the end sector of the log blocks and, thus, erase operations can be considerably reduced. In addition, the performance improves with the number of log blocks. The pattern E does not include any overwrite sequence for which O-FAST will benefit and, thus, the performance of FAST and O-FAST is identical.

Next, let us examine the pattern A and B, which are generated from digital cameras and thus contain both small random writes and large sequential writes. In both patterns, FAST consistently outperforms BAST, especially when the number of log blocks is equal to or less than 16. The performance gap in this range is mainly because of the reduction of erase count in the small random overwrites. As we mentioned in Section 2, the intelligent switch operation

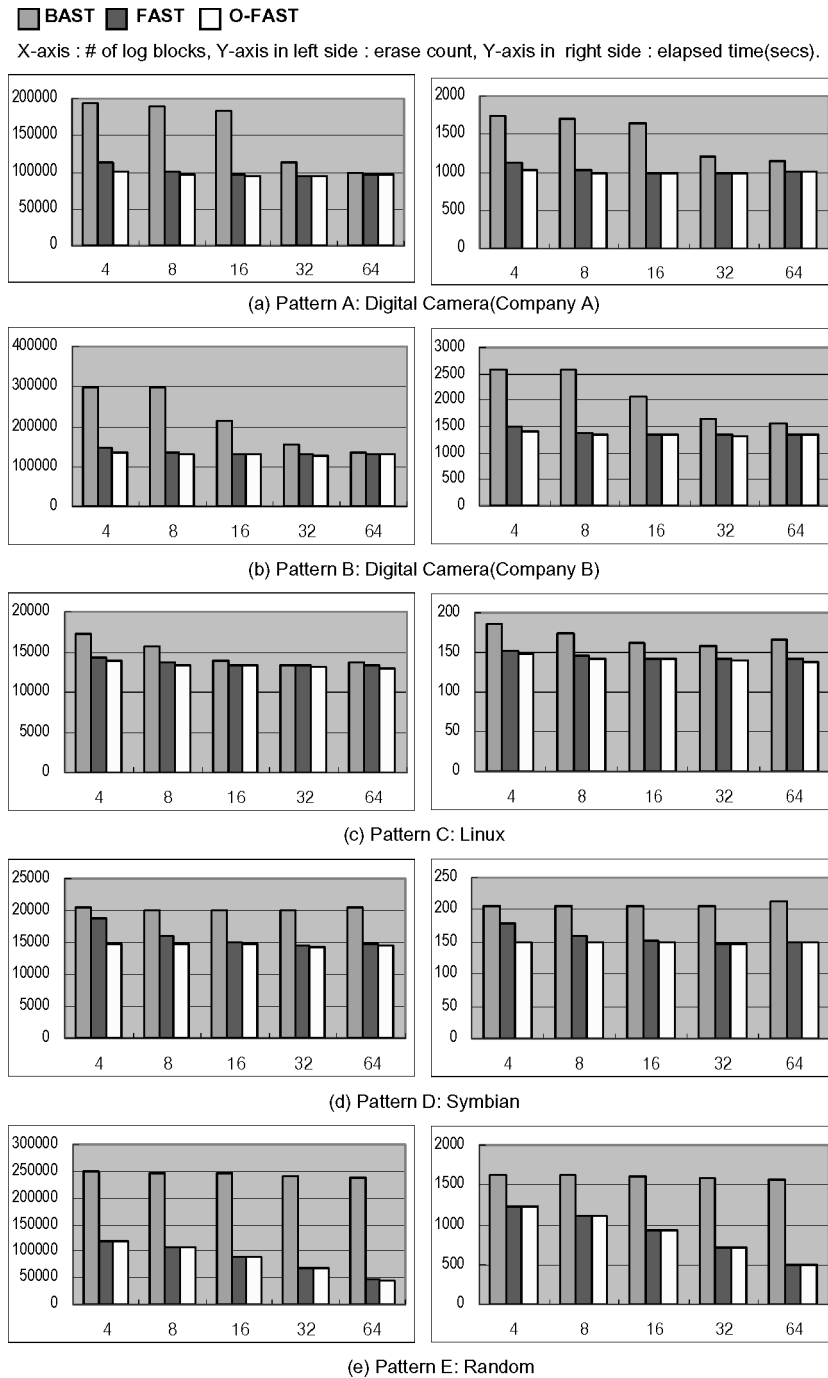


Fig. 10. Performance evaluation: results.

Table II. The Number of Switch Operations:
FAST versus BAST

# of log blocks	Pattern A		Pattern B	
	BAST	FAST	BAST	FAST
4	49,213	48,529	79,214	78,472
8	49,212	48,628	78,910	78,734
16	49,278	48,628	78,664	78,720
32	48,757	48,628	78,470	78,770
64	48,083	48,693	77,759	78,794

of BAST works well for large sequential writes. Our FAST scheme can also support sequential writes efficiently by introducing the SW log block. In order to check the efficiency of FAST against large sequential writes, we counted the number of switch operations both in BAST and FAST. As indicated in Table II, the number of switch operations in each scheme is almost same. From this result, we can conclude that the SW log block in FAST works well. As the number of log blocks increases, the performance of BAST approaches that of FAST. This is because the log block thrashing drastically diminishes as the number of log blocks becomes larger than the number of hot blocks and, therefore, the space utilization of each log block improves. With these patterns, O-FAST shows a small performance improvement over FAST. In fact, these patterns do not contain the overwrite patterns that O-FAST prefers.

Finally, let us consider patterns C and D, which contain many small random writes and small large sequential writes. In pattern D, the number of hot blocks at any point of time is about three or four. Therefore, regardless of the number of log blocks, the performance of BAST remains almost identical. In contrast, the performance of FAST improves with the number of log blocks. When the number of log blocks is less than or equal to eight, O-FAST shows some improvement over FAST. This is because small random writes in pattern D go perfectly well with the O-FAST preferred pattern. However, as the number of log blocks increases, the performance of FAST converges to that of O-FAST, which can be explained as follow: As the number of log blocks increases, the sectors in a victim block have more chance to be invalidated so the corresponding data blocks are less likely to be merged. In pattern C, the performance gap between FAST and O-FAST is negligible and the performance of BAST gets closer to FAST as the number of log blocks approaches 16.

One interesting result is that the performance of BAST depends upon the number of log blocks, especially when the number of hot blocks is large, while FAST shows excellent performance even with a small number of log blocks. In practice, the number of log blocks is generally limited to less than 10. With BAST, if the number of hot blocks gets larger than the number of log blocks, the performance degrades significantly. With FAST, a write operation can be done in any log block so that the performance does not much depend on the number of log blocks. As the number of log blocks increases, FAST can delay merge operations longer and thereby avoid some merge operations (as in the pattern D). This explains the performance improvement with the number of log blocks in FAST (especially in the pattern E).

5. CONCLUSIONS

In this paper, we proposed a novel FTL scheme, FAST, that outperforms the well-known log block scheme BAST. Its performance advantage mainly comes from the full associativity between the logical sectors and log blocks. By exploiting full associativity, FAST can avoid the log block thrashing phenomenon, delay merge operations as late as possible, and skip many unnecessary merge operations. These optimizations drastically reduce the number of expensive erase operations. Another advantage of FAST is that it guarantees a reasonable performance even with small number of log blocks and its performance improves gracefully with the number of log blocks. In fact, FAST with only 4 to 8 log blocks can provide the same performance as BAST with more than 30 log blocks.

In the future, we will devise some more optimization opportunities from full associativity. We will also explore what the theoretical performance optimum for flash memory is under a given workload. One good starting point is the concept of “ideal scheme” in Kim et al. [2002], which is defined as “a scheme that performs one erase operation for every n -sector write operation, where n is the number of sectors per block.” Finally, we would like to investigate the chasm between traditional file systems (i.e., Windows’ FAT, Unix’s file system) and Flash’s FTL, and its symptoms, such as file system aging. This work will be a corner-stone for flash-aware file system.

ACKNOWLEDGMENTS

We wish to thank Dr. Sang-Lyul Min and his FAST research group at Seoul National University for providing us with the several workloads used in this paper. We would also like to thank three anonymous reviewers who provided many useful comments that helped improve the quality of this paper.

REFERENCES

- BAN, A. 1995. Flash file system. United States Patent, No. 5,404,485, April.
- CHUNG, T. S., PARK, D. J., PARK, S. W., LEE, D. H., LEE, S. W., AND SONG, H. J. 2006. System software for flash memory: a survey. In *Proceedings of the 2006 IFIP International Conference on Embedded And Ubiquitous Computing (EUC 2006)*. (Aug.) Seoul, Korea.
- DOUGLIS, F., CACERES, R., KAASHOEK, M. F., LI, K., MARSH, B., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operation Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994, J. LEPREAU, Eds. Usenix Association, Berkeley, CA. 25–37.
- ESTAKHRI, P. AND IMAN, B. 1999. Moving sequential sectors within a block of information in a flash memory mass storage architecture, United States Patent, No. 5,930,815, July.
- GAL, E. AND TOLEDO, S. 2005. Algorithms and data structures for flash memories. *ACM Computing Surveys* 37, 138–163.
- HENNESSY, J. L. AND PATTERSON, D. A. 2003. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann, San Mateo, CA.
- KIM, B. S. AND LEE, G. Y. 2002. Method of driving remapping in flash memory and flash memory architecture suitable therefore, United States Patent, No. 6,381,176, April.
- KIM, J. S., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. K. 2002. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics* 48, 366–375.
- LAWTON, G. 2006. Improved flash memory grows in popularity. *IEEE Computer* 39, 1 (Jan.), 16–18.

- PAULSON, L. D. 2005. Will hard drivers finally stop shrinking? *IEEE Computer* 38, 5 (May), 14–16.
- SAMSUNG ELECTRONICS. 2005. Nand flash memory & smartmedia data book.
- SHINOHARA, T. 1999. Flash memory card with block memory address arrangement. United States Patent, No. 5,905,993, May.

Received April 2005; revised October 2005 and March 2006; accepted May 2006