

# FAST: An Efficient Flash Translation Layer for Flash Memory

Sang-Won Lee<sup>1</sup>, Won-Kyoung Choi<sup>2</sup>, Dong-Joo Park<sup>3</sup>

<sup>1</sup> School of Information and Communication Engineering, Sungkyunkwan University, Korea  
swlee@skku.edu

<sup>2</sup> Mobile Communication Division, Telecommunication Network Business, Samsung, Korea  
skkutop@hanmail.net

<sup>3</sup> School of Computing, Soongsil University, Korea  
djpark@ssu.ac.kr

**Abstract.** Flash memory is used at high speed as storage of personal information utilities, ubiquitous computing environments, mobile phones, electronic goods, etc. This is because flash memory has the characteristics of low electronic power, non-volatile storage, high performance, physical stability, portability, and so on. However, differently from hard disks, it has a weak point that overwrites on already written block of flash memory is impossible to be done. In order to make it possible, an erase operation on the written block should be performed before the overwrite, which lowers the performance of flash memory highly. In order to solve this problem, the flash memory controller maintains a system software module called the flash translation layer(FTL). In this paper, we propose an enhanced log block buffer FTL scheme, FAST(Fully Associative Sector Translation), which improves the page usability of each log block by fully associating sectors to be written by overwrites to the entire log blocks. We also show that our FAST scheme outperforms the previous log block buffer scheme.

**Keywords:** Operating Systems, Flash memory, FTL, Address translation, Associative mapping

## 1 Introduction

Flash memory is being rapidly deployed as data storage of PDAs, MP3 players, mobile phones, digital cameras, mainly because of its small size, low-power consumption, shock resistance, nonvolatile memory, and so on: [2], [6]. Moreover, compared to hard disk with the inevitable mechanical delay, such as seek time and rotational latency, in accessing data, flash memory provides fast uniform random access.

However, flash memory suffers from the write bandwidth problem. That is, a write operation is relatively slower than a read operation, and the write operation may have to be preceded by an erase because overwrite is not allowed in flash memory. Unfortunately, write operations can be performed in a unit of sector, while erase operations can be done only in a unit of block of which the size is significantly larger

than that of sector. Therefore, the cost of an erase operation is very costly, compared to that of read or write operations<sup>1</sup>. These inherent characteristics of flash memory reduce the write bandwidth, which is the performance bottleneck in flash memory-based mobile devices.

To relieve this performance bottleneck, it is very important to reduce the number of erase operations resulting from write operations. So a middleware called a flash translation layer (FTL) has been introduced between its upper-level file system and its lower-level flash memory: [3], [5], [6], [7]. The FTL plays a role of preparing empty locations in flash memory where to store data submitted by write operations. The well-designed FTL may soften the limitation of “erase-before-write” above. Among the various FTL schemes proposed so far, the log block scheme [6] is well-known for its good performance [1]. This scheme maintains a small size of log blocks in flash memory as temporary storage for overwrites. If a collision, namely an overwrite, happens at a certain location of flash memory, this scheme writes data to an empty location in a dedicated log block, not erasing the corresponding block. Since these log blocks play a role of cushions against overwrites, the log block scheme can significantly reduce the number of total erase operations against the same workload.

However, when a collision occurs, only a dedicated log block has to be used to store data. Due to the lack of log blocks, they have to be frequently replaced for other collisions. Unfortunately, most of the log blocks being replaced, usually has many unused empty locations. That is, the space usage rate of each log block is low. In this paper, we try to make such space usage rate high and therefore, improve the performance of write operations in the flash memory system.

In order to increase the space usage rate of the log blocks in the log block scheme, we see the role of log blocks with a different perspective. If we view the log blocks in [6] as “a cache for write operations”, each of logical sectors to be overwritten is mapped to only one log block, that is, the log block scheme writes each sector to its dedicated log block. In this respect, the address associativity between logical sectors and log blocks is of block-level. Thus, we call the log block scheme in [6] the Block-Associative Sector Translation (hereafter, BAST) scheme.

In this paper, we propose a novel FTL scheme, of which the main motivation is that the block-level associativity of the BAST scheme results in a poor write performance. If we make a degree of the associativity between logical sectors and log blocks higher, we can avoid the write performance degradation. In our scheme, a logical sector can be placed in any log block, so we call our scheme the Fully Associative Sector Translation (from now on, FAST) scheme. As in the computer architecture’s CPU cache realm [4], if we view the log blocks as a kind of cache for write operations and enlarge the associativity, we can avoid write miss ratio and therefore achieve a better FTL performance.

The remainder of this paper is organized as follows. The next section gives an overview of the BAST scheme and its disadvantages. A detailed description of our proposed FAST scheme, including its fundamental idea, is presented in the following section. Next, we compare the performance of our scheme with that of the BAST scheme. Finally, we conclude this paper and provide future work.

---

<sup>1</sup> Per-page costs of read, write, and erase operations are 25us, 300us, and 2ms, respectively, where the size of a page is 512Bytes [6].

## 2 The BAST Scheme

### 2.1 Overview

In this subsection, we will briefly review the BAST scheme[6]. The file system views flash memory as a set of logical sectors; that is, a hard-disk-like block device. So the write interface of FTL is defined as follows:  $write(lsn, sector)$ , which means “write a given  $sector$  to the location of the logical sector number  $lsn$ ”. On receiving a write request from the file system, the FTL finds a physical location in flash memory to write the sector as follows: it first calculates the logical block number (shortly, lbn) using the given  $lsn^2$  and then gets the physical block number (shortly, pbn) corresponding to lbn from the block-level mapping table<sup>3</sup>. Next, it calculates the offset in the found physical block at which the sector data will be written<sup>4</sup>. Finally, it writes the sector at the found offset in the data block (which is another representation of the physical block).

If the found offset in the data block was already occupied (that is, written) by one of the previous write operations, the FTL writes the given sector at the same offset in a free block allocated from the free block list. Next, all written sectors in the data block except the sector at the found offset are copied to the free block. Finally, the 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 an erase are required. A series of these-like operations is called the *merge* operation. To address this problem, many FTL techniques have been provided and of them, the BAST scheme is well-known as de facto the best FTL technique[1]. When collisions occur, the BAST scheme writes data to temporary storage, namely log blocks, which consequently reduces the number of merge operations. In the following, we describe the BAST scheme in detail using an example.

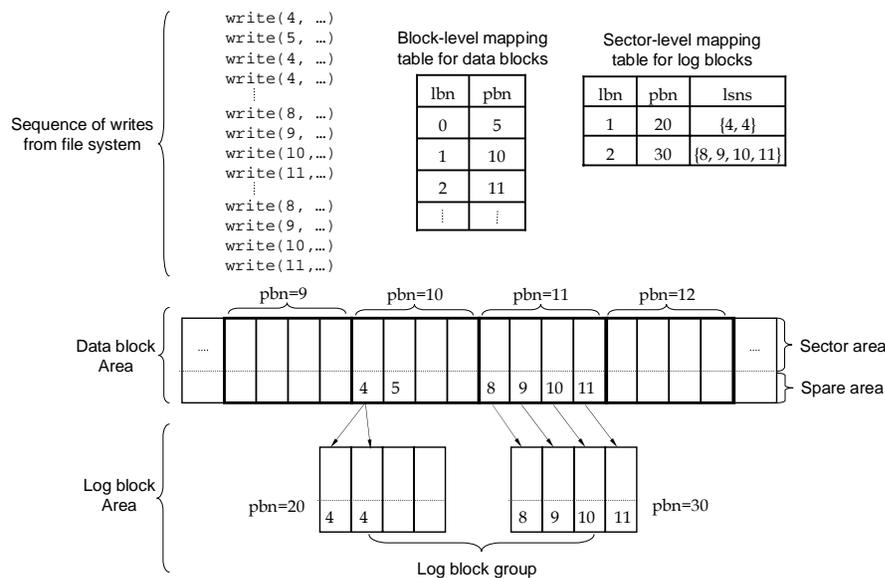
In Fig. 1, let the number of sectors per block be four and the number of log blocks two. In the figure, the upper-left part indicates a sequence of writes issued from the file system and the upper-center and the upper-right parts show the block-level and the page-level address mapping tables, respectively, which are usually maintained in SRAM area of the flash memory. If the first write operation is called in the figure, the BAST algorithm gets data block 10 from the block-level address mapping table using logical block 1 ( $= 4 \div 4$ ). Then it stores a given sector at offset 0 ( $= 4 \bmod 4$ ) in the data block 10. In case of the second write operation, the same thing as the first write operation is done. In case of the third write operation, a collision occurs in the data block 10, therefore the sector is written at the first offset in a log block (i.e., pbn=20) which is allocated to logical block 1 from the log block list. In case of the fourth write operation, the sector is directed to the next empty offset in the existing log block. The rest of the writes will make the second log block (i.e., pbn=30) and the sector-level address mapping table in the figure.

---

2  $lbn = (lsn \div \#sectors\_per\_block)$

3 The BAST scheme holds two block-level and page-level address mapping tables, which exist in the data block area and the log block area, respectively.

4  $offset = (lsn \bmod \#sectors\_per\_block)$



**Fig. 1.** Processing write operations in the BAST scheme.

A collision can occur in other data blocks, for example, data block 12 in Fig. 1. In this case, since there exists no more log block to be allocated, the BAST scheme selects, erase, and return one of the used log blocks. Before returning the victim log block, it needs to perform a merge operation in the original data block and it. That is, the up-to-date sectors between the two blocks are copied to a free block, which is then exchanged with the data block. At the same time, the block-level address mapping table has to be updated and the entry corresponding to the victim log block in the sector-level address mapping table has to be removed. After that, both the victim block and the data block are erased and one is returned to the log block list and the other to the free block list. One interesting fact is that, depending on the status of the victim log block, the merge operation can be optimized. For instance, in Fig. 1, all sectors in the log block of pbn=30 are sequentially written and the number of them equals the capacity of a block. In this case, any copy to the free block is not required, instead only the exchange of the victim log block with the data block is required. This operation is called especially the *switch* operation.

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

Here, let us briefly remind readers of the basics of the CPU cache. Because of the principle of locality in data access of computer programs(that is, reads and writes)[4], a small cache, combined with the memory hierarchy, can provide users with a very fast, very large, but very cheap memory system(Fig. 2(a)). In this respect, the log

blocks in the log block scheme can be also viewed as a kind of cache for the write operations(Fig. 2(b)). In the occurrence of collisions, by writing sectors in the log blocks, namely the cache, instead of original data blocks, we can complete the write operation much faster.

By the way, there is the associativity issue between memory and cache, as depicted in Fig. 2(a), that is, where can a memory block be placed in the cache? There are various associativities, including direct mapped, n-way associative, and fully associative[4]. With the “direct mapped” approach, a memory block can be placed only in a fixed cache block via a mathematical formula, while the “fully associative” approach allows a block to be placed anywhere in the cache. Borrowing these implications, we can say that the log block scheme takes the block-associative sector translation approach, in that overwrite operations for the sectors belonging to the same logical block can be allowed only in one log block allocated to this logical block(Fig. 2(c)).

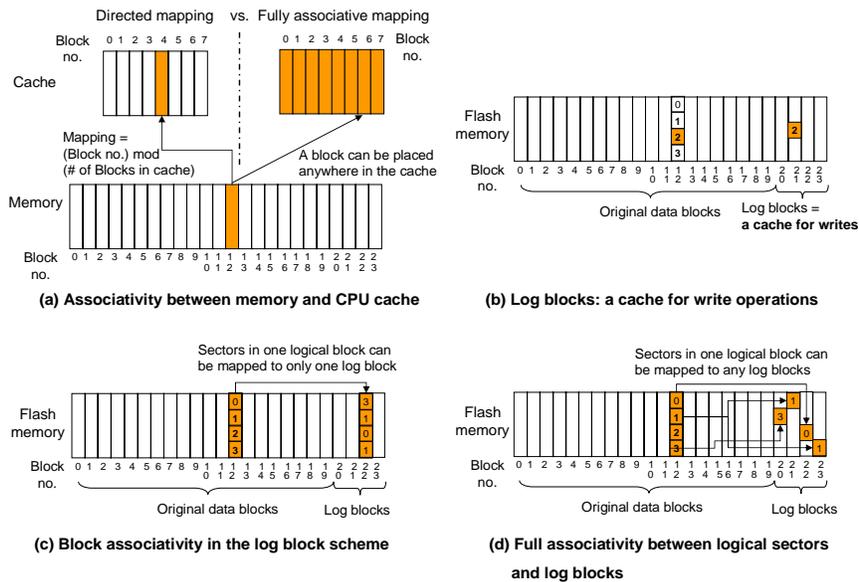


Fig. 2. CPU cache vs. log blocks: associativity.

### 2.3 Disadvantages

This block-level associativity of the BAST scheme gives rise to at least two performance problems. One of them is analogous to the high miss ratio in direct mapped associativity. If the cache, namely log blocks in flash memory, cannot accommodate all collisions during the execution of a program(especially, writes for hot blocks in flash memory), capacity misses will occur because of block thrashing[4]. In fact, the BAST scheme can suffer from similar thrashing problem.

For instance, assume that two blocks with four sectors per block each are allocated for the cache and that the write pattern is like (S0, S4, S8, S12, S0, S4, S8, S12) where  $S_i$  is the sector number for the write operation. For every writes from the second S0, the BAST scheme should replace one of the two log blocks, although the victim log block contains just one sector. We refer to this phenomenon as *log block thrashing*. These replacements will accompany costly merge operations.

Another performance problem stems from block-level associativity. Assume that under the same cache above, write operations for one hot logical block occur successively, e.g., the write pattern is like (S0, S2, S1, S3, S1, S0, S2, S3)<sup>5</sup>. In the BAST scheme, for every 4<sup>th</sup> 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, the log block scheme might work poorly against the continuous write operations for one hot block during a given time window.

### 3 FAST: A Log Buffer Scheme using Fully Associative Sector Translation

#### 3.1 Motivation

Based on the discussion in Section 2.3, we can think of a natural variation 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 blocks, which gives two performance optimization opportunities. The first one is to alleviate log block thrashing. Even if the number of different hot logical blocks in a given time window is greater than that of log blocks, the higher degree of associativity between logical sectors and log blocks can help reduce the miss ratio of empty sectors used for overwrites. This is very similar to the reduction in cache miss ratio when the associativity is increased. For example, the write pattern (S0, S4, S8, S12, S0, S4, S8, S12) in Section 2.3 does not require any block replacement under the fully associative approach, which thus does not yield any erase or merge operations.

The second (and more important) optimization opportunity is to reduce the number of merge operations which happen when the log blocks has no empty sector. Let us consider the write pattern (S0, S2, S1, S3, S1, S0, S2, S3) from Section 2.3 again. If we adopt fully associative mapping, only two log blocks are sufficient to place all sectors of the write pattern. So, we can avoid costly merge operations for every 4<sup>th</sup> write(which occur in the BAST scheme) and can also delay the merge time until all the empty sectors in the log blocks are wholly used. In summary, even under an environment that a certain logical block is very hot in a given time window, the full associativity approach inhibits the occurrence of merge operations.

---

<sup>5</sup> The hot logical block number is 0.

Even though these optimizations might seem to be very naïve, its performance impact is big. You can understand the advantages of full associativity between logical sectors and log blocks more clearly as you proceed with this paper.

### 3.2 Handling the Write Operations under the FAST Scheme

In this subsection, we explain how the FAST scheme handles the write operations issued from the file system. From the trace data of mobile devices like digital cameras, we can find that a large number of sectors are *sequentially* written by the file system, e.g., write pattern is like (S0, S1, S2, S3, ...) and in the middle of sequentially written sectors, sectors of a special sector area are written randomly and repeatedly. Generally, most of workloads traced from the flash memory systems consist of a large number of sequential writes and relatively small random overwrites. In order to cope with these write patterns, the log blocks are, in the FAST scheme, divided into two areas: one log block for sequential writes and the rest ones for random writes. The advantage of occupying one log block for sequential writes is that we can induce switch operations(in Section 2.1) cheaper than merge operations.

In order for a log block to be a target of the switch operation, it must satisfy the following two conditions: (1) the sector number(i.e., lsn) at the first offset in the log block must be divided by the number of sectors per block(say, 4), that is,  $lsn \bmod 4 = 0$  and (2) the log block must be filled up with the sectors which are sequentially written from the first to the last offsets. We try to place into the log block for sequential writes only the sectors which will satisfy both conditions potentially.<sup>6</sup> If a sector with the different logical block number shows up and satisfies the first condition, the existing sectors should be expelled from the log block for the new sector. However, in case of not satisfying the first condition, it will be placed into the log blocks for random writes. If, in the middle of filling up with the sectors, a desirable log block fails, in other words, not satisfying the second condition, we execute a merge operation between the failed log block and its original data block(to be explained later).

When collisions occur in data blocks, the corresponding sectors can be placed into any log blocks under the FAST scheme, as shown in Fig. 2. However, for the sake of managing the log blocks conveniently, we first fill up with the sectors into the first one of the log blocks for random writes, followed by the second one, and so on. If there is no more available empty space in the last log block, we select a victim log block in a round-robin fashion and then perform the merge operation between this victim block and its original data block. In turn, an erased log block will be returned to the log blocks for random writes.

We classify the merge operations into two types: for the log block for sequential writes and for the log blocks for random writes. In the former case, according to an appearance of the log block, the merge operation can be cheaper, that is, only one erase operation is required. For example, assume that the log block appears like “S4, -

---

<sup>6</sup> For instance, the write pattern(..., S4, S5, S6, S7, ...) will satisfy the two conditions. On the other hand, the write patterns (... , S6, S7, S8, S9, ...), (... , S4, S5, S5, ...), or (... , S4, S6, S7, ...) will not satisfy.

1, S6, -1” where -1 indicates “no sector written”. In this case, we copy corresponding sectors from its original data block into locations with -1’s and next exchange the updated log block with its original data block. Finally, we erase the original data block and use it as a log block for sequential write. Regarding the merge operation for random writes, it is more or less difficult. A victim log block can contain sectors whose logical block numbers differ from one another, for example, “S1, S8, S3, S9”. In this case, since two different groups of logical blocks, i.e., “S1, S3” and “S8, S9” exist in the victim, we have to perform two merge operations. Each merge operation is analogous to that of the BAST scheme, except that up-to-date sectors to be copied to a free block are searched from the overall log blocks for random writes. After that, for every up-to-date sector, not only its sector number but sector numbers of all the sectors older than it are changed into -1(meaning *invalid*) at their offsets of the corresponding log blocks. This helps us avoid merge operations when finding invalid marks in coming victim log blocks.

In order to lessen the cost of performing the two types of merge operations above, we maintain two sector-level address mapping tables, respectively, each of which records what sectors are written to what offsets in the log block(s).

## 4 Performance Evaluation

To make a comparison of FAST with BAST, we performed trace-driven and sample-driven simulations using three workloads in Table 1[6]. As shown in Table 1, the three workloads A, B, C include the various spectrum of the ratio of sequential writes to random writes, respectively. We use the number of erase operations as a performance metric. Fig. 3 shows the results of the simulations, with varying the number of log blocks. In the figure, the FAST scheme shows better performance than the BAST scheme, especially in the workload C(which contains mostly random writes). This is because the FAST scheme exploits fully the log block buffer for any type of workloads as mentioned in Section 3.

**Table 1.** Workload characteristics.

Workload	Description	# of writes
A	Traced from a notebook computer running Linux	398,000
B	Traced from a digital camera	3,144,800
C	Generated synthetically	150,000

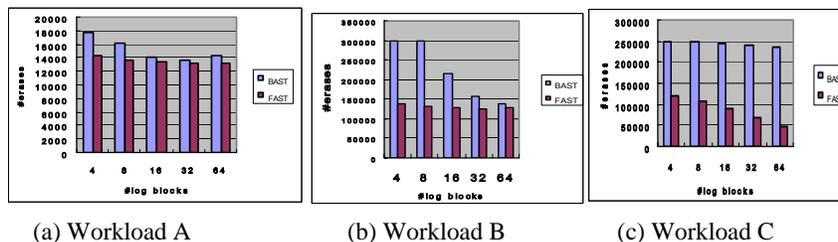


Fig. 3. Simulation results.

## 5 Conclusions

In this paper, we proposed a novel FTL scheme, called FAST, which outperforms the well-known log block scheme. Its performance advantage mainly comes from the full associativity between logical sectors and log blocks. By doing this, it can 1) avoid log block thrashing phenomenon, 2) delay merge operations to the maximum, and 3) skip many unnecessary merge operations. With only 4 to 8 log blocks in the FAST scheme, we can achieve the same performance result as the BAST scheme with more than 30 log blocks. In the future, we will exploit some more optimization opportunities from the full associativity. Also, we will investigate how to achieve the atomicity of file system operations when the power goes off unexpectedly[6]

**Acknowledgments.** This work was supported in part by MIC & IITA through IT Leading R&D Support Project, 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 partly by Seoul R&D Program(10660).

## References

1. Tae-Sun Chung, Dong-Joo Park, Sang-Won Park, Dong-Ho Lee, Sang-Won Lee, Ha-Joo Song: System Software for Flash Memory: A Survey, In Proceedings of the 2006 IFIP International Conference on Embedded And Ubiquitous Computing(2006)
2. F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, JA. A. Tauber: Storage Alternatives for Mobile Computers, In Proceedings of the 1<sup>st</sup> Symposium on Operation Systems Design and Implementation(1994)
3. Petro Estakhri, Berhanu Iman: Moving Sequential Sectors within A Block of Information in A Flash Memory Mass Storage Architecture, United States Patent, No. 5,930,815(1999)
4. John L. Hennessy, David A. Patterson: Computer Architecture: A Quantitative Approach (2nd ed.), Morgan Kaufmann(1996)
5. Bum Soo Kim, Gui Young Lee: Method of Driving Remapping in Flash Memory and Flash Memory Architecture Suitable Therefore, United States Patent, No. 6,381,176(2002)

6. Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho: A Space-Efficient Flash Translation Layer for CompactFlash Systems, IEEE Transactions on Consumer Electronics, Vol. 48, No. 2(2002)
7. Takayuki Shinohara: Flash Memory Card with Block Memory Address Arrangement, United States Patent, No. 5,905,993(1999)