# Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme

DAWOON JUNG
Korea Advanced Institute of Science and Technology
JEONG-UK KANG
Samsung Electronics Co.
HEESEUNG JO
Korea Advanced Institute of Science and Technology
and
JIN-SOO KIM and JOONWON LEE
Sungkyunkwan University

In NAND flash-based storage systems, an intermediate software layer called a Flash Translation Layer (FTL) is usually employed to hide the erase-before-write characteristics of NAND flash memory. We propose a novel superblock-based FTL scheme, which combines a set of adjacent logical blocks into a superblock. In the proposed Superblock FTL, superblocks are mapped at coarse granularity, while pages inside the superblock are mapped freely at fine granularity to any location in several physical blocks. To reduce extra storage and flash memory operations, the fine-grain mapping information is stored in the spare area of NAND flash memory. This hybrid address translation scheme has the flexibility provided by fine-grain address translation, while reducing the memory overhead to the level of coarse-grain address translation. Our experimental results show that the proposed FTL scheme significantly outperforms previous block-mapped FTL schemes with roughly the same memory overhead.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*

This work was supported by the Ministry of Knowledge Economy, Korea, under the (Information Technology Research Center) Support program supervised by the (Institute of Information Technology Advancement) (IITA-2009-C1090-0902-0020), and by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R01-2007-000-11832-0).
Authors' addresses: D. Jung, H. Jo, Computer Science Department, KAIST, Daejeon, Republic of Korea; email: {dwjung, heesn}@calab.kaist.ac.kr; J.-U. Kang, Memory Division, Samsung Electronics Co., Gyeonggi, Republic of Korea; email: ju.kang@samsung.com; J.-S. Kim, J. Lee, School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Gyeonggi-do 440-746, Republic of Korea; email: {jinsookim, joonwon}@skku.edu;

DOI 10.1145/1721695.1721706 http://doi.acm.org/10.1145/1721695.1721706

General Terms: Design, Performance

Additional Key Words and Phrases: FTL, NAND flash memory, storage system, hybrid address translation

**ACM Reference Format:**

Jung, D., Kang, J.-U., Jo, H., Kim, J.-S., and Lee, J. 2010. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 40 (March 2010), 41 pages.
DOI = 10.1145/1721695.1721706 http://doi.acm.org/10.1145/1721695.1721706

## 1. INTRODUCTION

Many mobile devices, including MP3 players, PDAs (personal digital assistants), PMPs (portable media players), high-resolution digital cameras and camcorders, and laptop computers, demand a large-capacity and high-performance storage system in order to store, retrieve, and process a large amount of multimedia data quickly. In mobile embedded devices, NAND flash memory is already becoming one of the most common storage medium because of its versatile features such as nonvolatility, solid-state reliability, low power consumption, small and lightweight form factor, shock resistance, and high cell densities [Douglis et al. 1994; Park et al. 2003; Inoue and Wong 2003]. At the same time, an increasing number of laptop computers are adopting NAND flash-based SSDs (solid-state disks) in place of hard disks.

Unlike conventional hard disks, NAND flash memory has a unique *erase-before-write* characteristic such that a page, which is the basic unit of read and write operations, should be erased before any new data can be written in the same location. The worse problem is that erase operations can only be performed on a block[1] basis, whose size is larger than a page by 32 to 128 times. Thus, emulating traditional block device interface on top of NAND flash memory necessitates an intermediate software layer called a *flash translation layer* (FTL) that addresses these characteristics [Kawaguchi et al. 1995; Intel Corp. 1998].

Typically, FTL redirects each write request to an empty location in NAND flash memory that has been erased in advance and manages an internal mapping table to record the address translation information from the logical page number to the physical location on flash memory. Although FTL gives an ability to update the same logical sector transparently, it adds extra flash memory operations to prepare empty locations and extra storage to maintain the mapping table. The amount of extra operations and storage required are drastically varied depending on the internal mapping scheme in FTL.

There is trade-off between the amount of extra storage and the number of extra flash operations. One can use a fine-grain address translation scheme in which each logical page can be located anywhere in flash memory, providing the best possible flexibility at the expense of extra storage for managing a huge amount of mapping information. As the capacity of NAND flash-based

[1]The "block" used in flash memory should not be confused with the unit of I/O used by the kernel. Unless otherwise stated explicitly, this article uses the term "block" to denote the unit of erase operation in flash memory.

storage increases, the extra storage required by the fine-grain address translation scheme actually imposes a serious cost problem in mass-market products [Kim et al. 2002]. On the other hand, it is possible to use a coarse-grain address translation scheme in which a series of consecutive logical pages, divided by the block size, are physically stored in the same block. The coarse-grain address translation scheme reduces the amount of extra storage as only the block-level mapping information needs to be maintained, but may cause more extra flash memory operations due to its inflexibility in dealing with write requests smaller than a block.

In this article, we propose a novel FTL, called *Superblock FTL*, which employs a hybrid address translation scheme for NAND flash memory. In the proposed scheme, we define a *superblock* as a set of adjacent logical blocks. Superblocks are mapped at coarse granularity, while pages inside a superblock are mapped freely at fine granularity to any location in a number of physical blocks allocated to the superblock. To reduce the amount of extra storage and extra flash memory operations, the fine-grain mapping information is stored in the spare area of NAND flash memory. This hybrid mapping scheme has the flexibility provided by the fine-grain address translation, while reducing the memory overhead to the level of coarse-grain address translation. The performance evaluation results show that our Superblock FTL scheme significantly reduces the flash memory management cost compared to previous coarse-grain FTL schemes with roughly the same memory overhead.

The rest of the article is organized as follows. Section 2 gives a brief overview of NAND flash memory and FTL. Section 3 describes the motivation of the proposed FTL. In Section 4, a detailed description of our Superblock FTL is presented. In Section 5, the performance of our scheme is extensively compared with previous schemes. Finally, we conclude the article in Section 6.

## 2. BACKGROUND AND RELATED WORK

In this section, we describe the characteristics of NAND flash memory and the differences among various NAND flash memory types. We also present a short overview of FTL and summarize related work.

### 2.1 Characteristics of NAND Flash Memory

A NAND flash memory chip is composed of a fixed number of *blocks*, where each block typically has 32 *pages*. Each page in turn consists of 512 bytes of main data area and 16 bytes of spare area. NAND flash memory does not support in-place update. Once a page is written, it should be erased before the subsequent write operation is performed on the same page. Since read and write (or program) operations are executed on a page basis while erase operations on a much larger block basis, NAND flash memory is sometimes called a write-once and bulk-erase medium.

The spare area in each page is often used to store out-of-band data such as a bad block indicator, page management information, and error correction code (ECC) to correct errors while reading and writing [Harari et al. 1997]. Note that the spare area can be read or written along with the main data area using a single read or write operation. Therefore, there is virtually no additional overhead to store/retrieve out-of-band data to/from the spare area.

Unlike hard disks or other semiconductor devices such as SRAMs and DRAMs, a write operation in flash memory requires a relatively long latency compared to a read operation. As the write operation usually accompanies the erase operation, the operational latency becomes even longer. Another limitation of NAND flash memory is that the number of program/erase cycles for a block is limited to about 100,000 to 1,000,000 times. Thus, the number of erase operations should be minimized not only to improve the overall performance but also to extend the lifetime of NAND flash memory.

Recently, a new type of NAND flash memory, called *large block NAND*, has been introduced in order to provide high density and high performance in bulk data transfer. In the large block NAND flash memory, a page consists of 2KB of main data area and 64 bytes of spare area, and a block has 64 pages. Note that a new programming restriction is added in the large block NAND flash memory; pages should be programmed in sequential order from page 0 to page 63 within a block. Random page address programming in a block is strictly prohibited by the specification [Samsung Elec. 2007]. Most of the latest NAND flash devices whose capacity is more than 1Gbits have the large block organization [Micron Technology Inc. 2005].

As semiconductor technology improves, multilevel cell (MLC) NAND flash memory has been introduced. In the previous single-level cell (SLC) NAND flash memory either with a small block or with a large block organization, each cell can represent only 1 bit. On the contrary, the voltage level of a single cell in MLC NAND flash memory is divided into four or more levels, with each cell representing more than 1 bit. This MLC technology allows for higher-capacity NAND flash memory with lower cost compared to the SLC technology. 2-bit MLC is already in mass production, while quad-bit MLC is expected to be available in the near future. In two-bit MLC NAND flash memory, the page size and the block size are doubled; each page has 4KB of main data area plus 128 bytes of spare area, and each block consists of 128 pages. Several packages of MLC NAND flash memory even uses the larger spare size, 218 bytes per 4KB of main data [Cooke 2007]. Although MLC NAND flash memory provides much higher capacity, several limitations should be noted. First, the read and write latency has been increased. Especially, a write operation sometimes takes 3 or 4 times longer than SLC NAND. Second, the bit error rate (BER) of MLC NAND is two orders of magnitude worse compared to SLC NAND, due to the reduced distance between adjacent voltage levels [Dan and Singler 2003]. This necessitates more powerful ECC to detect and correct multiple bit errors. Third, MLC NAND flash has smaller program/erase cycle limit (typically, around 10,000 cycles) due to the increased bit error rate. Finally, MLC NAND flash memory does not allow partial page programming. A whole page of MLC NAND flash memory should be programmed at once. It also has the same restriction as large block SLC NAND flash such that pages within a block should be programmed in sequential order.

Since the small block SLC NAND flash memory is being phased out in the market, we primarily focus on the large block SLC and MLC NAND flash

Table I. A Comparison of (Large Block) SLC and MLC NAND Flash Memory

| Characteristics | | SLC[1] | MLC[2] |
|---|---|---|---|
| Structure | Page size (KB) | 2 | 4 |
| | Spare size (Byte) | 64 | 128 |
| | Block size (KB) | 128 (64 pages) | 512 (128 pages) |
| Access time ($\mu$s) | NAND Flash data read time ($\mu$s/page) | 129.7 | 165.6 |
| | NAND Flash spare read time ($\mu$s/page) | 30.5 | 63.2 |
| | NAND Flash write time ($\mu$s/page) | 298.9 | 905.8 |
| | NAND Flash erase time ($\mu$s/block) | 1,998.7 | 1,500.0 |

[1]Based on Samsung K9F1G16U0M [Samsung Elec. 2003].
[2]Based on Samsung K9GAG08U0M [Samsung Elec. 2006].

memory in this article. Table I compares the structure and the access time of typical SLC and MLC NAND flash chips. The access time of SLC NAND flash memory is actually measured on Samsung K9F1G16U0M [Samsung Elec. 2003], and that of MLC NAND flash memory is estimated based on the datasheet of Samsung K9GAG08U0M [Samsung Elec. 2006].

## 2.2 Flash Translation Layer (FTL)

The main goal of FTL is to emulate the functionality of a normal block device with flash memory, hiding the presence of erase operation and the erase-before-write characteristics. To achieve this, FTL redirects each write request from the host to an empty location (free page) in flash memory that has been erased in advance and manages the mapping information internally. As a result of the write operation, the page storing the old data becomes invalid and the page in which the new data is written becomes a valid page. Among others, two particularly important functions of FTL are address translation and garbage collection.

The primary role of the address translation is to translate the logical sector number (e.g., logical block address [LBA]) of a request into a physical address that points to the corresponding page in flash memory. According to the granularity with which the mapping information is managed, FTLs are classified either as page-mapped [Ban 1995; Intel Corp. 1998] or as block-mapped [Ban 1999; Kim et al. 2002]. *Garbage collection* is the process that reclaims invalid pages scattered over the blocks by erasing appropriate blocks so that invalid pages are changed to free pages. Unless all the pages are invalid for the chosen block, a merge operation should be performed; before erasing the victim block, the valid pages in the block must be copied to some other blocks in order to prevent valid data from being lost.

A page-mapped FTL scheme is a fine-grain translation from a logical sector number to a physical block number and a physical page number, as shown in Figure 1(a). Since a logical sector can be mapped to a page in any location in NAND flash memory, the page-mapped FTL scheme permits more flexible storage management. However, the size of the mapping table becomes large in proportion to the total number of pages in NAND flash memory. Generally, the mapping table resides in RAM; therefore, it consumes a large amount of RAM.

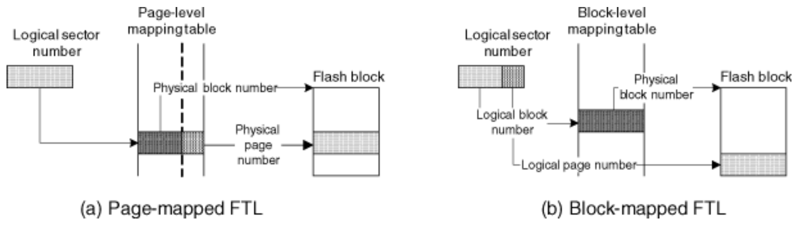


Fig. 1. Basic address translation schemes in FTL.

In a block-mapped FTL scheme, a logical sector number is divided into a logical block number and a logical page number, and then the logical block number is translated to a physical block number, as depicted in Figure 1(b). The logical page number helps to find the wanted page within the physical block. Unlike the page-mapped FTL scheme, each logical sector cannot be placed freely in flash memory under the block-mapped FTL scheme. Instead, a set of consecutive logical sectors should be stored in the same physical block. The size of the mapping table is only proportional to the total number of blocks in NAND flash memory. Therefore, the amount of RAM required by the block-mapped FTL scheme is significantly smaller compared to the page-mapped FTL scheme.

As the capacity of NAND flash-based storage increases, the large amount of RAM required by the page-mapped FTL scheme actually imposes a serious cost problem in mass-market products. For example, a Secure Digital (SD) card with a 4GB large block NAND flash memory chip requires 8MB of RAM for maintaining the mapping table with the page-mapped FTL scheme, while requiring only 128KB for the block-mapped FTL scheme. Thus, some variations of the block-mapped FTL scheme are widely used for NAND flash-based storage systems.

## 2.3 General Architecture of Block-Mapped FTLs

Generally, we can classify physical flash memory blocks into *D-blocks* (or data blocks) and *U-blocks* (or update blocks) according to their usage in block-mapped FTL schemes. D-blocks represent those blocks used to store user data. The total size of D-blocks serves as the effective storage space provided by FTL. A small number of U-blocks, which are invisible to users, are managed by FTL to handle the erase-before-write characteristics of NAND flash memory. When there is a write request to one of the pages and the write request cannot be accommodated in the corresponding D-block, FTL allocates a U-block and writes the fresh data into the U-block, invalidating the previous page in the D-block. Once a U-block is allocated, the subsequent write requests to the D-block can be redirected to the associated U-block. When the U-block itself becomes full, FTL can allocate another U-block or can generate a new D-block by merging the original D-block with the U-block. Although there are many different kinds of block-mapped FTLs, the difference largely comes from the way those D-blocks and U-blocks are managed.

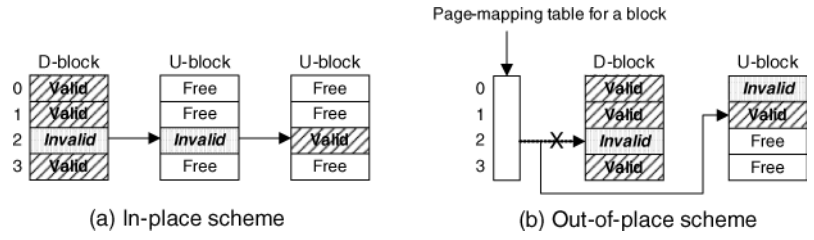


Fig. 2. Page-management schemes within a block.

Logical pages in a D-block or a U-block are organized either by in-place scheme or by out-of-place scheme. In the in-place scheme, the logical page number is always equal to the physical page number in the physical block; therefore, the logical page number is invariant during the address translation. In the out-of-place scheme, however, a page can be placed anywhere inside the physical block, requiring another page-level mapping information to find the exact location of the page.

Assume that the third page (logical page #2) in a D-block is updated twice in Figure 2. Under the in-place scheme (see Figure 2(a)), two extra U-blocks are allocated in order to write to the same location as the previous page. The in-place scheme simplifies the storage management, while other free pages in U-blocks may be wasted when only a part of pages is heavily updated. In addition, due to the sequential page programming restriction, using the in-place scheme is not always possible, especially in the large block SLC or MLC NAND flash memory.

In the out-of-place scheme (see Figure 2(b)), the logical page is written to any free page in a U-block and the page-mapping table for the block is modified to point to the newly written page. Although the out-of-place scheme is more flexible, the extra overhead is added to manage the second level of page-mapping table for each block. Thus, the out-of-place scheme is usually employed in a very limited way.

When all the available U-blocks are exhausted, a merge operation is invoked to generate a free U-block. During the merge operation, FTL selects a victim U-block and merges it with the corresponding D-block. According to the situations, the merge operation can be classified into *full*, *partial*, or *switch merge*, as illustrated in Figure 3. The full merge (see Figure 3(a)) is simple; it allocates a free block that is erased beforehand, and then copies the most up-to-date pages (we call them valid pages), either from the D-block or from the U-block, into the free block. After copying all the valid pages, the free block becomes the D-block and the former D-block and the U-block are erased. Therefore, a single full merge requires read and write operations as many as the number of valid pages in the merged blocks and two erase operations.

Partial and switch merges are special cases of the full merge operation. The partial merge takes place when all the valid pages in the D-block can be copied to the rest of the U-block. As shown in Figure 3(b), the partial merge copies only the valid pages in the D-block and one erase operation can be saved compared to the full merge. On the other hand, if all the pages in the D-block are already

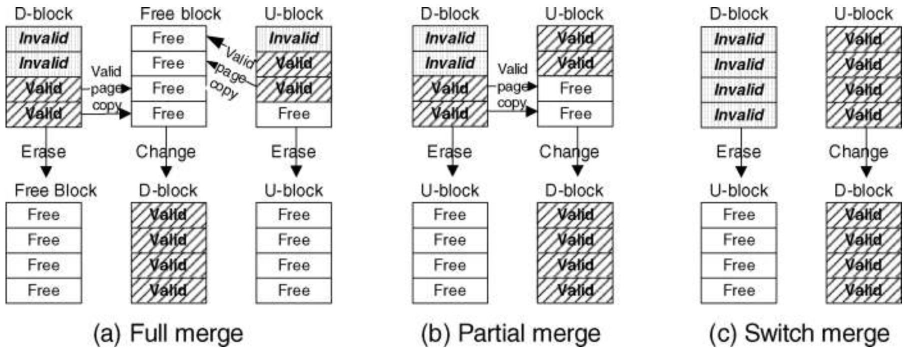


Fig. 3. The types of merge operations.

invalidated, we can simply switch a U-block to a new D-block and erase the old D-block. This case is called the switch merge (see Figure 3(c)). The switch merge requires only one erase operation without any valid page copy and hence is the most efficient case among merge operations. The switch merge typically occurs when the whole pages in a block are sequentially updated. This is the storage access pattern commonly found in many file systems when they attempt to store large multimedia or archive files.

The performance of block-mapped FTLs significantly depends on how to organize D-blocks and U-blocks, and on how to select victim U-blocks during merge operations. We note that the performance degradation in FTL is mainly caused by copying valid pages and performing erase operations to make free blocks during merge operations.

## 2.4 Related Work

As described in Section 2.2, FTLs can be classified into page-mapped FTLs and block-mapped FTLs according to the mapping granularity. Various schemes have been proposed to improve the performance of FTLs.

DAC [Chiang et al. 1998] is one of the most popular page-mapped FTL schemes. The key idea of DAC is to cluster pages that have data with a similar update frequency into the same block. DAC logically partitions flash memory into several regions. To classify pages into separate regions according to its update frequency, data migrate between regions. When data are updated, they are promoted to the upper region, while data are demoted back to the lower region when the associated block is erased. In this way, DAC tries to maximize the chance that pages in a block become invalidated together in a certain period. During garbage collection, the block can be reclaimed with small overhead because the number of valid pages to be copied remains fairly small. Owing to the fine-grain address translation, DAC can flexibly cluster data into a block. However, a large amount of RAM is required to maintain fine-grain mapping information. In addition, finding a victim block for garbage collection is a time-consuming job, as DAC examines all the block information to find the victim block. Due to these drawbacks, page-mapped FTLs can be used, if any, only for a small-sized flash memory.

For high-capacity flash memory storage systems, block-mapped FTLs are widely used mainly due to their relatively small memory requirement. Ban [1995] has proposed the *replacement block scheme* based on the concept of a replacement block. In this scheme, U-blocks are called replacement blocks, and both D-blocks and U-blocks are organized by the in-place scheme. The operation of the replacement block scheme is similar to the example shown in Figure 2(a). When there is a write request, it allocates a U-block if the write cannot be accommodated in the existing D-block and U-blocks. During garbage collection, the D-block, which has the largest number of U-blocks, is selected as a victim, and all the valid pages are copied into the last U-block. The last U-block then becomes a new D-block. Since the pages are always merged into the last U-block, only the partial or the switch merge is performed. As noted in the previous section, the replacement block scheme exhibits poor storage utilization especially when only some of the pages are frequently updated. Moreover, this scheme is not suitable for the recent flash memory, where pages in a block cannot be programmed in random order.

Kim et al. [2002] have suggested the *log block scheme* that uses U-blocks as logging blocks. The log block scheme logs the changes of the data stored in a D-block into a U-block until the U-block becomes full. In the log block scheme, D-blocks are organized by the in-place scheme, while U-blocks by the out-of-place scheme in order to overcome the disadvantage of the replacement block scheme. If there is a write request, the log block scheme writes the data into the U-block sequentially and maintains the separate page-level mapping information only for U-blocks. Since only the small number of U-blocks is used by FTL, the additional mapping overhead can be kept low. When all the U-blocks are used, some U-blocks are merged with the corresponding D-blocks to secure a new free U-block. As D-blocks are managed by the in-place scheme, the full merge may happen in order to change from the out-of-place scheme to the in-place scheme. In addition, the utilization of U-blocks can still be low, since even a single page update of a D-block necessitates a whole U-block similar to the replacement block scheme.

To solve the problem of the log block scheme, Lee et al. [2007] have recently proposed the fully associative sector translation (FAST) scheme. In FAST, a U-block is shared by all the D-blocks, and every write request is logged into the current U-block. This effectively improves the storage utilization of U-blocks and delays the merge operation much longer. However, the full merge may be performed more frequently than the previous schemes, since a single log block contains pages that belong to several D-blocks. To alleviate this problem, FAST uses a special U-block, called *sequential log block*, and handles sequential writes in a special way.

Wu and Kuo [2006] have proposed AFTL, an FTL scheme that dynamically and adaptively switches between fine-grain and coarse-grain mapping granularities. The main objective of AFTL is to provide fast address translation with a small amount of memory for large-capacity flash memory. AFTL achieves this goal by using a page-level mapping table in memory for hot pages in U-blocks. The rest of the pages are managed by a coarse-grain mapping table similar to the replacement block scheme. When a U-block is fully written, valid pages in the block are considered as hot and managed in the page-level mapping table. If the page-level mapping table is full, the least recently used mapping table entries are evicted, and those pages are merged with the corresponding D-blocks. AFTL allocates more than one U-block into a D-block to delay merge operations and uses free pages in D-blocks for further updates. However, the latter optimization is only possible in the small block NAND flash. In addition, in most cases, it is not possible to exploit free pages in D-blocks, since all the storage space is written during `format` or `mkfs` to check bad blocks. Overall, AFTL can be viewed as a variant of the replacement block scheme with a small, fixed in-memory cache for address translation of hot pages.

Recently, Park et al. [2007, 2008] have studied N+K mapping scheme, which is in part influenced by our earlier work [Kang et al. 2006]. Similar to our Superblock FTL, the N+K mapping scheme organizes $N$ blocks into a group and allocates up to $K$ U-blocks to each group. The distinction is that pages in D-blocks are stored by the in-place manner in the N+K mapping scheme, while we organize D-blocks by the out-of-place manner to maximize flexibility. In fact, the main goal of their study is not to propose an efficient FTL but to propose an effective design space exploration methodology for the optimal values of $N$ and $K$, which show the best performance in the given workload. Although the methodology is useful to understand the characteristics of the target workload, the result cannot be used directly as they do not consider the memory requirement or the management overhead for the mapping information, especially when $N$ or $K$ becomes large.

## 3. MOTIVATION

In this article, we propose Superblock FTL that combines the adjacent logical blocks into a superblock. In our Superblock FTL, pages inside a superblock can be freely mapped at page granularity to several physical blocks allocated for the superblock. This section elaborates the motivation of our work.

### 3.1 Analysis of the Merge Cost

Let $\mathbf{W} =< w_1, w_2, \ldots, w_i, \ldots, w_n >$ be a trace of write requests issued from the host. $\mathbf{W}$ is a time-ordered list of logical page numbers of length $n$, where $w_i$ denotes the $i$-th logical page number written. Let $C_w$ be the cost to write a page in NAND flash memory and $\mathit{WriteCost}(\mathbf{W})$ be the total cost of writing the given trace $\mathbf{W}$. Apparently, $\mathit{WriteCost}(\mathbf{W})$ can be represented as the summation of the time to write $|\mathbf{W}|$ pages and the cost associated with merge operations, $\mathit{MergeCost}(\mathbf{W})$, as follows.

$$\mathit{WriteCost}(\mathbf{W}) = C_w \cdot |\mathbf{W}| + \mathit{MergeCost}(\mathbf{W}) \tag{1}$$

Equation (1) shows the unique performance characteristics of NAND flash memory compared to hard disks. Since NAND flash memory is a solid-state device that has no seek time, each write operation has a constant cost, namely $C_w$, regardless of the location of the sector written. On the other hand, the additional merge cost is unavoidable due to the erase-before-write characteristics of physical medium. From Equation (1), we can see that the write performance of

FTL mostly depends on the efficiency of garbage collection. Thus, reducing the garbage collection overhead has been a primary goal in designing FTLs [Chiang et al. 1999; Kim et al. 2002; Lee et al. 2007; Park et al. 2008].

The total merge cost, $\mathit{MergeCost}\,(\mathbf{W})$, is the summation of the individual cost to make a free page for each write request $w_i$, which we refer to as $\mathit{MergeCost}_{ind}\,(w_i)$. $\mathit{MergeCost}_{ind}\,(w_i)$ includes the time to erase blocks (referred to as $C_{erase}(w_i)$) for making free blocks, as well as the time to copy valid pages from victim U-blocks or D-blocks to a new block (referred to as $C_{copy}(w_i)$). Note that if there is already a free page in the D-block or the associated U-block, the write request does not incur any additional overhead. Using $\mathit{MergeCost}_{ind}\,(w_i)$, $\mathit{MergeCost}\,(\mathbf{W})$ is given by

$$\mathit{MergeCost}\,(\mathbf{W}) = \sum_{i=1}^{|\mathbf{W}|} \mathit{MergeCost}_{ind}\,(w_i) \tag{2}$$

where

$$\mathit{MergeCost}_{ind}(w_i) = \begin{cases} C_{erase}\,(w_i) + C_{copy}\,(w_i), & \text{if a merge operation occurs} \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Assume that $\mathit{MergeCount}\,(\mathbf{W})$ denotes the total number of merge operations performed for the given trace $\mathbf{W}$. Many FTLs try to minimize $\mathit{MergeCount}\,(\mathbf{W})$, that is, the number of occurrences that a new U-block is allocated, by sharing the existing U-block among multiple write requests. The higher the storage utilization of U-blocks grows, the lower the frequency of merge operations tends to be. For example, unlike the replacement block scheme, the log block scheme uses the out-of-place scheme for U-blocks so that several updates to the same logical block can be absorbed in the existing U-block regardless of the logical page number. FAST goes one step further to increase the utilization of U-blocks by allowing any updates to be logged in the current U-block.

However, Equation (3) tells us that it is equally important to reduce the cost of the individual merge operation, $\mathit{MergeCost}_{ind}\,(w_i)$, to improve the overall FTL performance. One way to reduce $\mathit{MergeCost}_{ind}\,(w_i)$ is to raise the chance of partial and switch merge operations while preventing the full merge operation from taking place as much as possible, so as to reduce $C_{erase}(w_i)$ and $C_{copy}(w_i)$. In fact, two factors, $\mathit{MergeCount}\,(\mathbf{W})$ and $\mathit{MergeCost}_{ind}\,(w_i)$, are dependent on each other; hence, both factors should be considered carefully in designing FTLs to minimize the overall merge cost.

## 3.2 Exploiting Block-Level Locality

Typical storage access patterns exhibit both the block-level spatial locality and the block-level temporal locality. This observation has already been mentioned in several literature including Ruemmler and Wilkes [1993] and Chang and Kuo [2005].

The *block-level spatial locality* represents that the pages in the adjacent logical blocks are likely to be updated in the near future. The block-level spatial locality appears when two or more adjacent logical blocks are allocated by file systems to the same file or to the same metadata such as FATs (file allocation tables), directories, i-nodes, and bitmaps. In this case, if several adjacent logical blocks share a U-block, the storage utilization of U-blocks will increase.

In our Superblock FTL, we define the superblock as a set of adjacent logical blocks that share D-blocks and U-blocks. The advantage of using the superblock is that we can exploit the block-level spatial locality to increase the storage utilization of U-blocks, while controlling the degree of sharing by adjusting the superblock size. We define the degree of sharing for a physical block as the number of logical blocks to which the pages, stored in the given physical block, belong.

FAST achieves the best storage utilization for U-blocks by logging every write request to a single log block regardless of the logical block number of the target page. Hence, in the worst case, the degree of sharing in FAST is identical to the number of pages within a block. As noted in Section 2.4, this tends to increase the merge cost. The log block scheme is another extreme case, where the degree of sharing is always limited to one. In the log block scheme, the block-level spatial locality is not exploited at all, which curtails the utilization of the log block. Therefore, we can notice that it is necessary to increase the degree of sharing for better storage utilization, but not too much, so that the merge cost can be kept low. We will explain the basic idea to reduce the merge cost in Section 3.3.

On the other hand, the *block-level temporal locality* indicates that the pages in the same logical block are likely to be updated again in the near future. The log block used in the log block scheme is essentially the mechanism to capture the block-level temporal locality, by redirecting the update requests to the same logical block into the associated log block.

Our Superblock FTL exploits the block-level temporal locality by allocating more than one U-block to each superblock, hoping that the other pages in the superblock will be updated soon by the subsequent write requests to the same superblock. Usually, the merge operation is delayed until there is a shortage of U-blocks for other superblocks. At the time the merge is required for a superblock, there will be several U-blocks allocated for the superblock, many pages of which are already invalidated due to the block-level temporal locality. This effectively increases the opportunity of performing the partial or switch merge operation instead of the costly full merge operation.

## 3.3 Hot–Cold Separation Using Page-Level Mapping inside a Superblock

We call a page *hot* if the page is relatively frequently updated compared to other pages in the logical block. Otherwise, the page is *cold*. In the previous block-mapped FTL schemes, the merge cost usually increases when both hot pages and cold pages are stored in the same logical block together. To illustrate the problem, consider the situation shown in Figure 4. In this example, we assume that the number of physical pages per block is four and only a single U-block is available in the system. For a given write trace $\mathbf{W} = < P0, P0, P5, P6, P8 >$, four hot pages, namely $P0$, $P5$, $P6$, and $P8$, are being updated.

In the log block scheme shown in Figure 4(a), each logical block possesses a different U-block. To update $P5$ at $w_3$, we have to merge D-block 0 with U-block

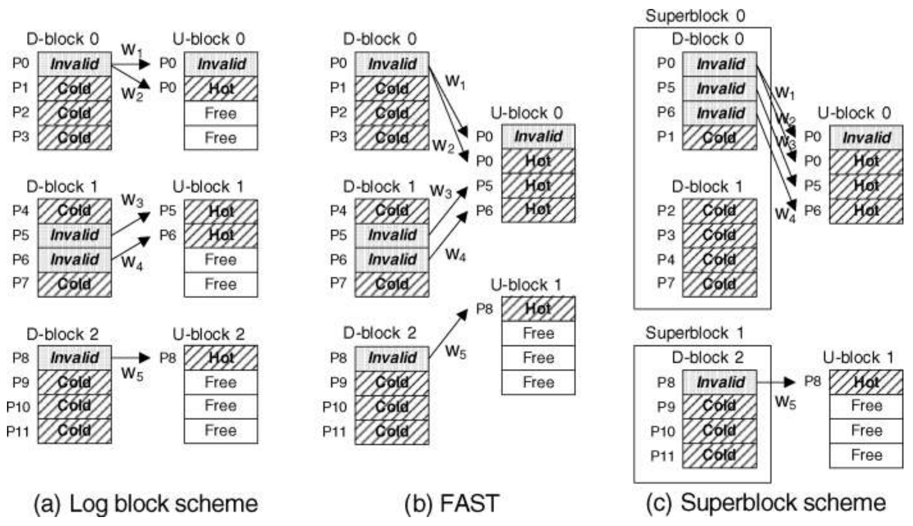


Fig. 4. Examples of handling write requests in block-mapped FTLs, when $\mathbf{W} = < P0, P0, P5, P6, P8 >$.

0 to make a free block. Since there are not enough free pages left in U-block 0, a full merge operation should be performed, resulting in two erase and four page copy operations. At $w_5$, another block merge between D-block 1 and U-block 1 is required to give a free block to D-block 2. Although there is enough space in U-block 1 to copy valid pages from D-block 1 ($P4$ and $P7$), a partial merge operation cannot be taken, since the pages in D-blocks need to be arranged by the in-place scheme. As a result, the merge cost at the moment is again two erase and four page copy operations, and the total merge cost is given by four erase and eight page copy operations.

In FAST (see Figure 4(b)), the block merge is not carried out until $P8$ is updated at $w_5$ because all the previous update requests can be handled using U-block 0. However, two full merge operations are still required at $w_5$ as U-block 0 has the pages that belong to both D-block 0 and D-block 1. FAST first merges D-block 0 with U-block 0 to generate the new D-block 0, and then merges D-block 1 with U-block 0 again for the new D-block 1. Thus, the merge cost in FAST is given by three erase and eight page copy operations. Compared to the log block scheme, $MergeCost(\mathbf{W})$ is reduced by one erase operation, and $MergeCount(\mathbf{W})$ is also decreased from two to one.

Under our Superblock FTL scheme, assume that D-block 0 and D-block 1 are grouped together to form a superblock. If we can place all the hot pages to D-block 0 and all the cold pages to D-block 1, as presented in Figure 4(c), we can reduce not only the merge count but also the individual merge cost. Since logical blocks in a superblock share a U-block, the merge operation is delayed until $w_5$ as in FAST. However, as all the invalid pages are now stored in D-block 0, only a single full merge operation is required between D-block 0 and U-block 0, resulting in two erase and four page copy operations for the total merge cost.

Table II. The Characteristics of Previous Work and Superblock FTL

| | | Replacement Block Scheme [Ban 1995] | Log Block Scheme [Kim et al. 2002] | FAST [Lee et al. 2007] | Superblock Scheme |
|---|---|---|---|---|---|
| D-blocks | Terminology | Data blocks | Data blocks | Data blocks | D-blocks |
| | Management scheme | In-place | In-place | In-place | Out-of-place |
| | Max. degree of sharing | 1 | 1 | 1 | $N$ (the superblock size) |
| U-blocks | Terminology | Replacement blocks | Log blocks | Random and Sequential log blocks | U-blocks |
| | Management scheme | In-place | Out-of-place | Out-of-place | Out-of-place |
| | Max. degree of sharing | 1 | 1 | 32 or 64 (the number of pages in a block) | $N$ (the superblock size) |
| Block merge | Frequency | high | middle | low | low |
| | Average Cost | middle | middle | high | low |

The key observation is that if we can dynamically arrange the pages into a physical block according to their hotness, we can reduce the merge cost. It is already pointed out in the previous study on page-mapped FTL schemes that the performance of FTL can be improved by relocating hot pages and cold pages to different physical blocks [Chiang et al. 1999]. Unfortunately, this technique could not be used for traditional block-mapped FTL schemes, since the page was not able to move outside the associated block boundary.

In our Superblock FTL, we still use the block mapping at the superblock level, but we allow logical pages within a superblock to be freely relocated in one of the allocated D-blocks and U-blocks by maintaining the page-level mapping information within the superblock. During merge operations, we try to separate hot pages from cold pages and put them into different D-blocks (details will be explained in Section 4.3).

We summarize the characteristics of previous work and the proposed Superblock FTL in Table II.

## 4. SUPERBLOCK FTL

In this section, we describe the design and implementation of the proposed Superblock FTL in detail.

### 4.1 Overall Architecture

The basic idea behind Superblock FTL is to map pages that belong to $N$ logical blocks to any location in up to $N + M$ physical blocks. $N$ indicates the number of logical blocks composing a single superblock, which is, in most cases, equal to the number of D-blocks allocated to the superblock. $M$ denotes the maximum number of U-blocks that can be attached to each superblock.

We construct a superblock by combining several adjacent logical blocks in order to utilize the block-level spatial locality. For example, if the superblock size is four, four logical blocks whose logical block numbers are 0, 1, 2, and 3 form a superblock 0. When a write request arrives for any page in the superblock, Superblock FTL allocates an empty U-block and logs the write request in the first page of the U-block.

A U-block is exclusively used by the associated superblock to exploit both the block-level temporal locality and the block-level spatial locality. Once a U-block is allocated to a superblock, the subsequent write requests to the superblock are logged in the U-block sequentially. This out-of-place scheme is suitable for use with the large block SLC or MLC NAND flash memory, in which pages should be programmed in sequential order from the first page to the last page within a block. When there are no more free pages in the U-block, another U-block is allocated for the superblock, as mentioned in Section 3.2. Some of these U-blocks are eventually turned into D-blocks during garbage collection. Further details on manipulating D-blocks and U-blocks are given in Section 4.3.

In order to make Superblock FTL useful, we need to consider the following: (i) how to maintain the mapping information compactly and efficiently, and (ii) how to intelligently merge D-block and U-blocks to reduce $MergeCount(\mathbf{W})$ as well as the individual merge cost $MergeCost_{ind}(w_i)$. In the following text, we attempt to answer these questions in detail.

## 4.2 Address Translation

4.2.1 *Hybrid Mapping with Three-Level Mapping Table.* Since Superblock FTL utilizes the page-level mapping inside a superblock, the pages belonging to $N$ logical blocks can be distributed anywhere in up to $N + M$ physical blocks. The page-level mapping information should be capable of covering all pages in $N + M$ blocks. In addition, the mapping information is frequently accessed by various FTL operations. Therefore, maintaining the address translation information efficiently and compactly is a challenging issue.

The simplest way of keeping such information in block-mapped FTLs is to store LBA in the spare area of the corresponding page and then to scan all spare areas in a block to find the particular page or to build the mapping information on demand. In Superblock FTL, however, since the size of a superblock is much bigger than that of a flash memory block, on-demand scanning incurs longer latency than in other block-mapped FTLs.

Instead, we use spare areas more aggressively to record the entire page-mapping information of each superblock. When user data are written in the main data area, the up-to-date page-mapping information is also stored simultaneously in the spare area of the same physical page. In this way, we can avoid any additional overhead in terms of space and flash operations. Although this strategy looks simple, it is not straightforward to implement, since the spare area is limited in its size.

To make the mapping information fit into the limited size of the spare area, we organize the page-mapping table in three levels, as shown in Figure 5. The overall architecture resembles the page table structure used in modern CPUs

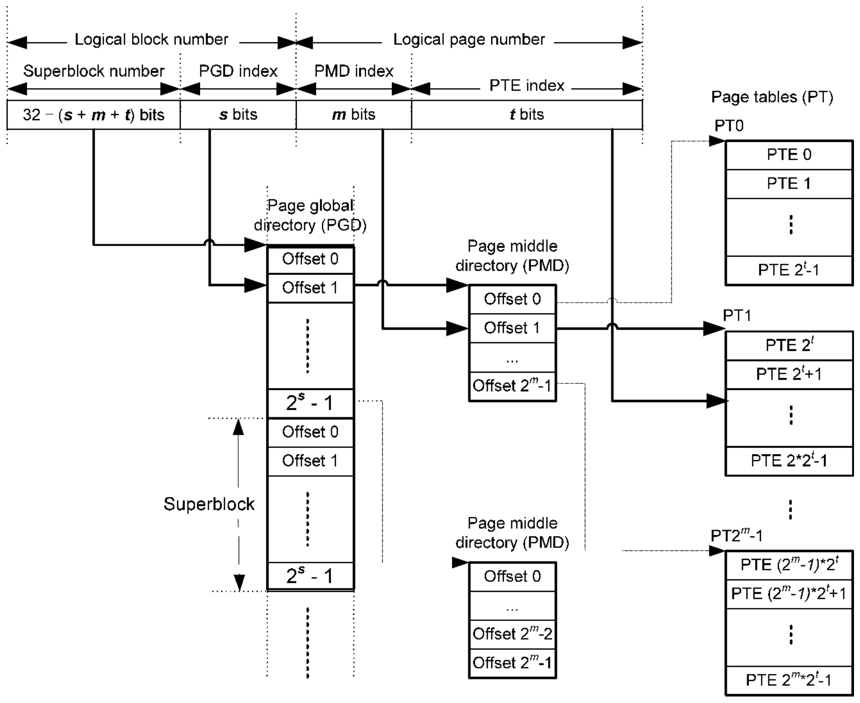


Fig. 5. The address translation in Superblock FTL with three-level page-mapping table.

for implementing virtual memory system. The first-level page table is the page global directory (PGD) indexed using the superblock number and PGD index. When the superblock size is $N = 2^s$, PGD index is low $s$ bits of the logical block number. Each entry of PGD points to a page middle directory (PMD) that holds $2^m$ entries. Each PMD entry, in turn, points to the location of one of $2^m$ page tables (PTs), whose entry (page table entry [PTE]) contains the physical block number and the physical page number of the wanted data. Using the high $m$ bits of the logical page number, which we call PMD index, we retrieve the location of PT from PMD and find the final PTE using the remaining $t$ bits of the logical page number, PTE index. Note that $2^{(m+t)}$ should be equal to the number of pages in a block.

The role of PMD is to locate the up-to-date position of each PT. The location of the up-to-date PMD is kept track of by PGD. While PGD is stored in main memory, PMD and PTs are saved in the spare area of NAND flash memory. Since the number of entries in PGD is equal to the number of logical blocks, the memory overhead for PGD is comparable to other block-mapped FTL schemes.

The rationale for this three-level mapping structure can be briefly explained as follows. Let us consider a hypothetical situation where the spare area is large enough to hold all the $2^{(m+t)}$ PTEs for a given logical block. In this case, the latest address translation information for any block can be retrieved from the spare area of the most recently written page in the block. All we have to

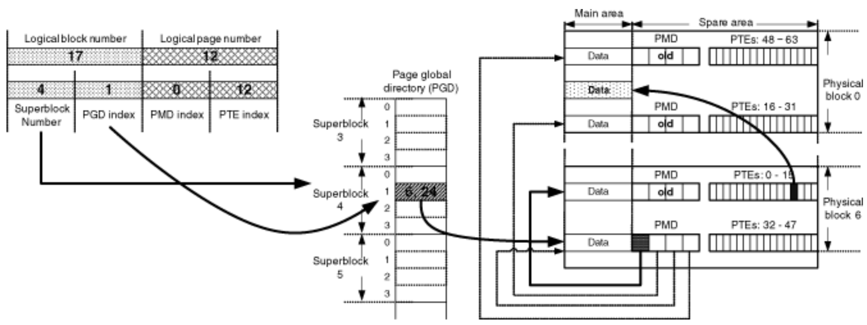


Fig. 6. An example of the address translation in Superblock FTL.

do is to let PGD keep track of the physical location of the last page written for each logical block.

In reality, however, the size of the usable spare area is far smaller than what is required to store $2^{(m+t)}$ PTEs. Our basic idea is to split $2^{(m+t)}$ PTEs into a set of $2^m$ PTs and to record only the affected PT into the spare area of the updated page. Consequently, the latest $2^{(m+t)}$ PTEs are distributed over $2^m$ different pages inside a superblock, which necessitates another level of data structure, that is, PMD, to maintain the current locations of $2^m$ PTs. Since one of PT and PMD need to be changed whenever a page is updated, we write PMD and the corresponding PT into the spare area along with the main data. Thus, Superblock FTL works as long as there is a space for storing PMD and one PT in the spare area. There is only one valid PMD in each logical block, whose location is maintained by PGD.

4.2.2 *Address Translation for SLC NAND Flash.* Figure 6 illustrates an example of address translation performed in Superblock FTL on large block SLC NAND flash memory. For the large block SLC NAND flash memory, the whole page table is divided into four separate PTs (i.e., $m = 2$) due to the space limitation of the spare area within a single page. Since a block consists of 64 pages, each PT has 16 PTEs (i.e., $t = 4$).

Suppose that we would like to find the physical address corresponding to the logical address whose logical block number is 17 and the logical page number is 12. The logical block number is divided into the superblock number 4 and PGD index 1, and the logical page number is split into PMD index 0 and PTE index 12. As shown in Figure 6, we find the latest PMD for the logical block 17 from PGD using the superblock number 4 and PGD index 1. Once PMD is read from the spare area, we extract the first entry from PMD to find the location of PT0. PT0 holds PTEs from PTE0 to PTE15, and the location of data can be found by reading PTE12 from PT0.

When a logical page is updated, the up-to-date page-mapping information is also saved in the spare area of the same physical page. For instance, suppose that the logical page that we find in the previous example is updated. In this case, PTE12 is modified to point to the location that the logical page will be written, and the first PMD entry is also changed to locate the same physical

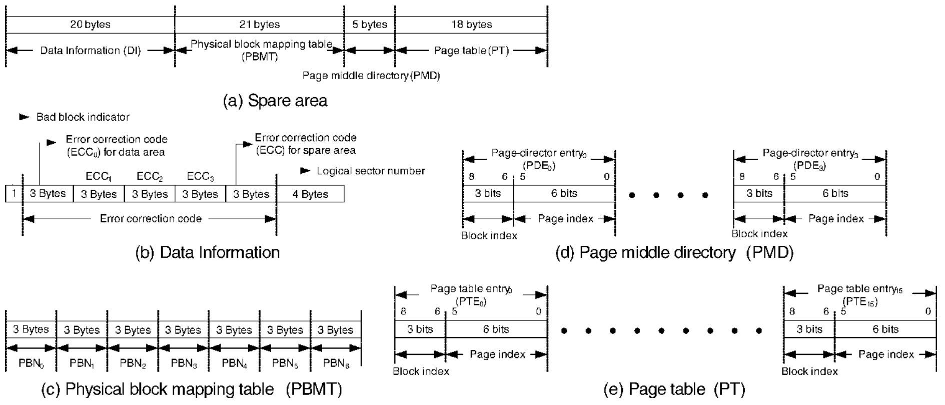

20 bytes
21 bytes
5 bytes
18 bytes
Data Information (DI)
Physical block mapping table (PBMT)
Page table (PT)
Page middle directory (PMD)
(a) Spare area
Bad block indicator
Error correction code ($ECC_0$) for data area
Error correction code (ECC) for spare area
Logical sector number
$ECC_1$
$ECC_2$
$ECC_3$
1
3 Bytes
4 Bytes
Error correction code
(b) Data Information
Page-director entry$_0$ ($PDE_0$)
Page-director entry$_3$ ($PDE_3$)
8 6 5 0
3 bits
6 bits
Block index
Page index
(d) Page middle directory (PMD)
$PBN_0$ $PBN_1$ $PBN_2$ $PBN_3$ $PBN_4$ $PBN_5$ $PBN_6$
(c) Physical block mapping table (PBMT)
Page table entry$_0$ ($PTE_0$)
Page table entry$_{15}$ ($PTE_{15}$)
(e) Page table (PT)

Fig. 7. The spare area format for SLC NAND flash memory in Superblock FTL for recording the page-mapping information.

page, since it now has the new PT0. After the page is written with the modified PMD and PT0, the second PGD entry is changed to point to a new location. As the up-to-date PMD and the corresponding PT is stored in flash memory whenever a page is updated, we can guarantee that each entry of PMD and PT always point to the valid page.

Since our FTL should read PMD and the corresponding PT from flash memory every time when the FTL read, write, or copy a page, we introduce a *map cache* to reduce the number of flash read operations. A map cache entry consists of PMD and one of the associated four PTs that are used to record the page-mapping information of a single logical block. The number of map cache entries is fixed and we manage those entries based on a least recently used (LRU) replacement policy. This cache mechanism is similar to those used in the log block scheme and FAST. Our experimental results show that the small number of map cache entries works quite well (see Section 5.7).

Figure 7 depicts the overall layout of the spare area for large block SLC NAND flash memory. The spare area is divided into four sections: data information (DI), physical block mapping table (PBMT), PMD, and PT, as presented in Figure 7(a). DI consists of a bad block indicator, 15 bytes of error correction code (ECC), and a logical sector number (see Figure 7(b)). The logical sector number in DI is typically used for recovery. PBMT is an array of seven physical block numbers, as shown in Figure 7(c). Each PMD has four page directory entries (PDEs) for locating four PTs (see Figure 7(d)), and each PT consists of 16 PTEs (see Figure 7(e)).

In principle, each PDE or PTE needs to point to a physical location of a page in flash memory, where the location is identified by the physical block number and the page offset inside the block. Allowing every PDE or PTE to specify the physical block number redundantly is not only wasteful but also impossible due to the limited size of the spare area. Instead, we adopt an indirect mapping to accommodate the whole information in the spare area. In our Superblock FTL, PBMT has an array of actual physical block numbers allocated for the

Table III. The Spare Area Formats for SLC and MLC NAND Flash Memory in Superblock FTL

| | | For SLC NAND ($m = 2, t = 4$) | | | For MLC NAND ($m = 3, t = 4$) | | |
|---|---|---|---|---|---|---|---|
| | Field Name | Count | Unit Size (bit) | Size (byte) | Count | Unit Size (bit) | Size (byte) |
| DI | Bad block indicator | 1 | 8 | 2 | 1 | 16 | 2 |
| | Logical sector number | 1 | 32 | 4 | 1 | 32 | 4 |
| | ECC for data area | 4 | 24 | 12 | 8 | 56 | 56 |
| | ECC for spare area | 1 | 24 | 3 | 1 | 48 | 6 |
| PBMT | Physical block number | 7 | 24 | 21 | 9 | 24 | 27 |
| PMD | PDE Block index | 4 | 3 | 2 | 8 | 4 | 4 |
| | PDE Page index | 4 | 6 | 3 | 8 | 7 | 7 |
| PT | PTE block index | 16 | 3 | 6 | 16 | 4 | 8 |
| | PTE Page index | 16 | 6 | 12 | 16 | 7 | 14 |
| Total | | | | 64 | | | 128 |

superblock, and the block index in PDE or PTE is used to retrieve the proper physical block number from PBMT. Then, the page index is used to identify the target physical page in the block.

Since there are 64 pages in a physical block of the large block NAND flash memory, 6 bits of page index in PDE or PTE are sufficient to locate any physical page in a block. The block index in PDE or PTE is 3 bits, which can indicate one of eight physical blocks. There are only seven physical block numbers in PBMT due to space limitation, and the eighth index has a special meaning. If the block index is specified as 7, it points out that the target physical block number is the same as that of the upper-level data structure; in case of PDE, it represents that the target PT is on the same physical block with PMD. For PTE, it denotes that the target page is on the same physical block with PT. This indirect mapping scheme for physical block numbers implies that the total number of D-blocks and U-blocks that can be allocated to a superblock is limited to eight in our current implementation.

4.2.3 *Address Translation for MLC NAND Flash.* Since the architectural characteristics of MLC NAND flash memory are different from those of SLC NAND, the format of the mapping table needs to be adjusted. There are some notable differences that affect the mapping table structure. First, the number of pages in a block is increased from 64 to 128; hence, each PTE entry requires 7 bits for page index. This also doubles the total number of PTEs managed by PTs. Second, a larger portion of the spare area in MLC NAND should be reserved for ECC. SLC NAND usually employs ECC for 1-bit error correction among 512-byte data. On the contrary, most MLC NAND manufacturers recommend to use ECC that can correct at least 4-bit errors per 512-byte data. The ECC capable of this requires 62 bytes for both 4-KB data and 128-byte spare area, implying almost half of the spare area is dedicated to ECC.

Table III compares the spare area formats for SLC and MLC NAND flash memory with 4-bit ECCs under Superblock FTL. For MLC NAND flash memory, the data information is comprised of 2-byte bad block indicator, 4-byte logical sector number, 56-byte ECC for 4-KB data, and 6-byte ECC for 128-byte spare

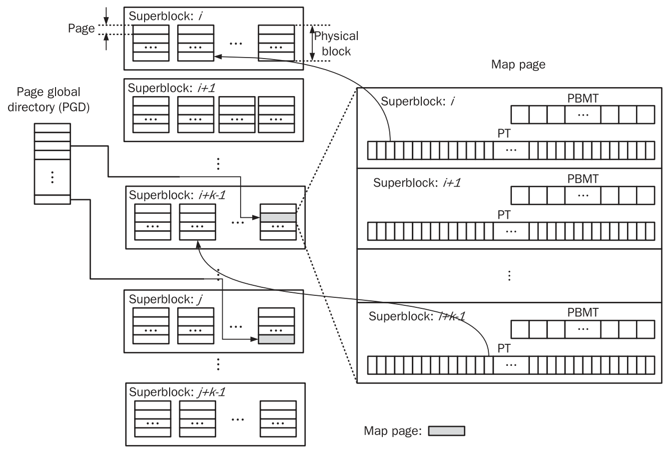


Fig. 8. Map page structure.

area itself. Due to the increased size of ECC, a single spare area can only accommodate 16 PTEs (i.e., $t = 4$), and the whole PTEs should be divided into 8 PTs. Accordingly, each PMD has 8 entries (i.e., $m = 3$). All the remaining space is assigned to PBMT. PBMT has 9 entries, which allows to allocate up to 10 physical blocks to a superblock.

MLC NAND flash, however, often requires stronger ECC for higher reliability. Protecting 4-KB data area with 8-bit/512-byte BCH error correction requires 104 bytes of spare area (out of 128 bytes). This implies that almost all the spare area needs to be dedicated to ECC, making it difficult to utilize the spare area for keeping the page-level mapping information. Recently, manufacturers are introducing a new MLC NAND flash architecture, which expands the spare area size to 218 bytes (per 4-KB data) to accommodate stronger ECC as well as other management information [Cooke 2007]. With this type of NAND flash memory, we can still exploit the spare areas for storing the page-level mapping information.

However, conventional MLC NAND flash, which has a standard 128 bytes of spare, is widely employed. In this case, Superblock FTL may adopt an alternative strategy, which keeps the page-level mapping information in a separate page called a *map page*. For each physical block, Superblock FTL reserves at least one map page to store the page-level mapping information. The structure of the map page is quite similar to that of the spare area except that the mapping information is organized in two levels without PMDs. As illustrated in Figure 8, each PGD entry keeps track of the location of each map page. When a page is accessed, Superblock FTL looks up the corresponding map page in PGD. Whenever a new map page is written to flash memory, PGD is also modified to indicate the up-to-date location of the map page.

Since the size of the map page is much larger than that of the spare area, a single map page can contain PBMTs and PTs for multiple superblocks. If the whole page is dedicated to a single superblock (i.e., $k = 1$ in Figure 8), the superblock size can grow up to 15 ($N = 15$) and each superblock may have up to

17 U-blocks ($M = 17$). On the other hand, if the superblock size is four ($N = 4$) with the maximum number of U-blocks being restricted to four ($M = 4$), a single map page can hold the mapping information for six superblocks ($k = 6$ in Figure 8).

Unlike mapping information stored in spare area, writing a map page requires an additional flash write operation because a map page is stored in a separate page. This connotes that updating a map page increases the write traffic of NAND flash memory. To alleviate the traffic, Superblock FTL caches map pages like the map cache. When the page-level mapping information is modified, the cached page buffers the update instead of writing the map page to flash. The page is flushed to flash memory either when the associated superblock is fully merged or when the dirty map page needs to be evicted from the cache. Superblock FTL even uses the cache to accelerate accesses to map pages. When a map page is fetched from flash memory, it is cached in memory.

## 4.3 Merge Operation

We need an intelligent merge mechanism in order to reduce the number of erase operations and valid page copies, which are the main sources of performance degradation. A merge operation is invoked when one of the following situations occur; (i) when a block is completely invalidated, (ii) when a free block cannot be assigned to a superblock because the total number of D-blocks and U-blocks allocated for the superblock reaches the limit, and (iii) when there is no free block to allocate. For the first situation, the block is immediately reclaimed (switch merge). Whenever a page is invalidated, we update the number of valid pages in the corresponding block and see if it is the last valid page in the block. Since no pages are copied in this situation, the merge requires only one erase operation.

In the second situation, a merge operation is needed even though free blocks are available. This is because the number of blocks composing a superblock is limited to 8 for SLC NAND and 10 for MLC NAND in our current implementation. The merge process for this case is illustrated in Figure 10. First, we load all the mapping information for the given superblock *sb* by reading spare areas (line 3) and find a victim block that has the minimum number of valid pages (line 5). The actual reclamation of the victim block is done by the algorithm shown in Figure 9. The valid pages in the victim block are first copied into the last U-block hoping for the partial merge. In case it becomes full, we allocate a new free block and copy the remaining pages to the block. When all the valid pages are copied, the victim block is erased. This routine is repeated until two free PBMT entries are generated. This particular number is determined by the simulation, as it shows the lowest merge cost.

The final situation is when no free block is available. In this situation, a victim superblock is selected and then U-blocks and D-blocks in the superblock are merged to make free blocks. To select the victim superblock, we maintain an LRU list of superblocks that have at least one U-block. Once the victim is selected, all D-blocks and U-blocks that belong to the superblock are merged together so that the superblock is composed of D-blocks only. The detailed merge process is illustrated in Figure 11.

**Algorithm 1.** Copying valid pages

```
1: procedure SimpleCopyAndCompact($sb, src$)
2:     $dest := GetLastUBlock(sb)$                              ▷ Gets the last allocated U-block of $sb$
3:     $ret := 1$
4:     $RemoveFromSB(src)$                                      ▷ Removes $src$ from $sb$
5:     for all valid page $p$ in block $src$ do
6:         if $dest$ has no free page then
7:             $ret := 0$
8:             $dest := AllocReservedBlock()$
9:             $AddToSB(sb, dest)$                              ▷ Adds $dest$ to $sb$
10:            $SetLastUBlock(sb, dest)$                        ▷ Sets $dest$ as the last U-Block of $sb$
11:        end if
12:        $ValidPageCopy(p, dest)$
13:    end for
14:    $Erase(src)$
15:    if $ret = 0$ then
16:        $AddToReservedBlock(src)$
17:    end if
18:    return $ret$
19: end procedure
```

Fig. 9. Copy and compact procedure. Valid pages in the *src* block are copied to free space, and *src* is erased.

**Algorithm 2.** Merge some - Case 2

```
1: procedure MergeSome($sb$)
2:     $nFreed := 0$
3:     $LoadAllMapTable(sb)$                                    ▷ Gets all block information of $sb$
4:     repeat
5:         $victim := FindVictimGreedy(sb)$  ▷ Finds a block which has the minimum number of valid pages
6:         $nFreed := nFreed + SimpleCopyAndCompact(sb, victim)$
7:     until $nFreed < 2$
8: end procedure
```

Fig. 10. Merge algorithm invoked when the number of D-blocks and U-blocks of a superblock reaches the physical limit.

The first step is to load all the mapping information for the victim superblock *sb* (line 3). Then, we classify blocks that belong to the superblock into hot blocks and cold blocks (line 7). The hot/cold information is stored by using 1 bit for each PBMT entry. D-blocks are initially marked as cold and U-blocks as hot when they are assigned to a superblock.

In the second step, all valid pages in hot blocks are packed in a new free block (lines 8 through 12). If the last U-block does not have any invalid pages, free pages in the U-block can be used to copy valid pages from other hot blocks, eventually being converted to a D-block. Otherwise, free pages in the last U-block are discarded as it cannot be a D-block (lines 4 through 6). When all valid pages of a hot block are migrated, the block is erased for further allocation. If free pages in the newly allocated block are exhausted, another free block is allocated to pack hot blocks. After compacting hot blocks, pages in cold blocks are also packed in the same manner (lines 13 through 17).

**Algorithm 3.** Merge all - Case 3

```
 1: procedure MERGEALL($sb$)
 2:     $dest := GetLastUBlock(sb)$
 3:     $LoadAllMapTable(sb)$                        ▷ Gets all block information of $sb$
 4:     if $dest$ has one or more invalid pages then
 5:         $InvalidateAllFreePages(dest)$           ▷ Abandons free pages of $dest$
 6:     end if
 7:     $< H, C >:= ClassifyHotColdBlocks(sb)$
 8:     for all hot block $h$ in $H$ do
 9:         if $h$ has no free page and has invalid pages then
10:             $SimpleCopyAndCompact(sb, h)$
11:         end if
12:     end for
13:     for all cold block $c$ in $C$ do
14:         if $c$ has no free page and has invalid pages then
15:             $SimpleCopyAndCompact(sb, c)$
16:         end if
17:     end for
18:     for all $block$ in $sb$ do
19:         $SetCold(block)$                         ▷ Sets all blocks of $sb$ into D-blocks
20:     end for
21: end procedure
```

Fig. 11. Merge algorithm invoked when no free block is available in the system.

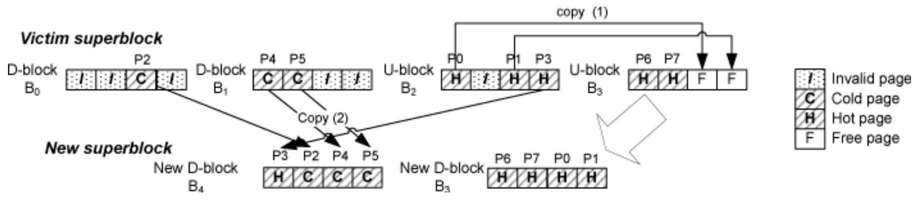


Fig. 12. An example merge operation in Superblock FTL.

Finally, all the blocks in the superblock become D-blocks and marked as cold (lines 18 through 20). The merge process completes by removing the victim superblock from the LRU list. Let us assume that the victim superblock was composed of $N + \alpha$ blocks, where $N$ is the superblock size and $\alpha$ denotes the number of U-blocks associated with the superblock. As a result of MERGEALL algorithm, we restructure the superblock to have only $N$ D-blocks, reclaiming $\alpha$ free blocks from the superblock.

Our merge algorithm is based on the observation that U-blocks tend to have relatively hot pages, while D-blocks have cold pages. Clustering hot pages into the same block is desirable for future merge operations, since it is highly probable that hot pages are updated again [Chiang et al. 1999]. Likewise, cold pages will not be updated for a long time or even never be updated. Therefore, clustering cold pages in the same block is helpful for improving efficiency of upcoming merge operations.

Figure 12 shows an example merge operation performed in Superblock FTL. In the example, the superblock size is two, and two additional update blocks are assigned to the superblock. As specified in Figure 11, hot pages in $B_2$ are first copied into free pages in the last U-block $B_3$ (Step (1)), and then pages in cold blocks are clustered into a new block $B_4$ (Step (2)). We can see that hot pages and cold pages are separated from each other simply by rearranging the location of each valid page inside the superblock during merge operations.

Unlike other block-mapped FTLs, the distinction between D-blocks and U-blocks is somewhat confusing in Superblock FTL, as each page can be located in any $N + M$ physical blocks inside a superblock. Let $N_{d_i}$ and $N_{u_i}$ represent the number of D-blocks and the number of U-blocks for the given superblock $i$, respectively. Basically, whenever a new free block is allocated to the superblock $i$, it is regarded as a U-block, increasing $N_{u_i}$ by 1. These U-blocks are eventually turned into D-blocks during garbage collection.

In the first and the second merge situations, $N_{d_i}$ and/or $N_{u_i}$ may be decreased depending on the type of victim block. If the final number of physical blocks is same as $N$, that is, $N_{d_i} + N_{u_i} = N$ as a result of the merge operation, all U-blocks are promoted to D-blocks. Similarly, in the final merge situation where $N_{d_i} + N_{u_i}$ blocks are compacted into $N$ physical blocks, all the resulting blocks are set to D-blocks.

To improve response time of Superblock FTL, merge operations can be performed in the background. When there is no pending requests, Superblock FTL can explicitly trigger merge operation to produce free space. This distributes the merge cost into idle periods, hiding the cost during handling normal requests. The background merge is especially effective when the intervals between bursts of requests are long. Numerous studies have already mentioned the background merge as a sole research problem. Chang et al. [2004] proposed a garbage collection scheme for real-time systems, which performs merge operations in the background as a separate thread. Choudhuri and Givargis [2008b, 2008a] also suggested partial block cleaning for guaranteeing performance, which divides the merge process into several steps and schedules each step among normal requests. We can address the background merge in the same way. However, this background merge is applicable not only to Superblock FTL but also to all other FTLs and flash file systems, and most literatures have addressed only FTL architectures and algorithms. In addition, a number of distinct issues, such as how many blocks to be reclaimed and when background merge is performed, arise when we implement background merge. These issues are dependent on the workload characteristics rather than the architectures of FTLs. Considering these, background merge can be regarded as an orthogonal issue in designing FTLs. Thus, we do not disscuss the details of the background merge in this article and leave it as future work.

## 4.4 Reliability Issues

In designing an FTL, another important concern is to ensure reliability in any harsh environment. In particular, FTLs usually confront a situation where the system is suddenly shut down due to system failure or instant power outage. The ramification of this is that FTLs lose all the information stored in in-memory data structures. Especially, missing PGD is critical to our Superblock FTL, as it cannot work properly without the top-level mapping information.

Another problem is that the metadata of FTL may remain inconsistent after sudden power failure because any FTL operation in progress is aborted. For example, allocating a new U-block to a superblock involves two steps: (i) the U-block is removed from the free block list, and (ii) the U-block is added in the mapping table for the superblock. If the system had been down somewhere between Step (i) and (ii), the U-block would be neither on the free block list nor allocated to any superblock. Therefore, after power-on, it is required to identify the exact location interrupted and to roll back or redo the failed operation correctly for maintaining consistency among FTL metadata.

Note that this is not a unique problem that arises only in Superblock FTL. Every FTL faces the same problem, and we can apply a similar approach that is used in other FTLs. In particular, taking a snapshot is one of the common techniques used in various flash-based storage systems [Kim et al. 2002; Yim et al. 2005; Lim and Park 2006; Bityutskiy 2005], and here, we briefly outline the recovery strategy from sudden power-off based on snapshots.

When Superblock FTL is normally shut down, a full snapshot is written into flash memory. The full snapshot consists of the following information: PGD, the free block list, the U-block list, and the valid page counter for each block. Therefore, Superblock FTL can restore the up-to-date states from the full snapshot after FTL is gracefully turned off. As the full snapshot contains PGD, the size of the full snapshot is proportional to the capacity of flash memory, and the snapshot can occupy several hundreds of pages. Superblock FTL tries to reduce the number of written pages for the snapshot by writing only the modified pages. In the worst case, however, the entire snapshot should be written. If 32GB of SLC NAND flash is employed, the PGD size is 768KB and the size of the other information including the list of free blocks and U-blocks and the valid page counters is about 216KB. In total, the full snapshot size is 984KB. In case of MLC NAND flash with map pages, the full snapshot size is 70KB due to the reduced the PGD size (8KB) and the increased block size. As a result, maximum 147.1ms and 16.3ms are required for 32GB of SLC NAND and MLC NAND, respectively.

The full snapshot size is a linear function of the flash memory capacity. This implies that taking the full snapshot with a huge capacity of flash memory takes a large amount of time. Even though 1 TB of MLC NAND flash is used, however, the full snapshsot size is 2.2MB,[2] and the time taken to write the full snapshot becomes 507.2ms, still being less than 1 second.

Whenever a block is allocated, we take a partial snapshot, which contains the address of the allocated block and the list of written pages with their LBAs and original page addresses after the last allocation. The partial snapshot is organized to fit into a single physical page. Even when no block is allocated, the partial snapshot is taken if the list of updated pages exceeds the partial snapshot size. After writing one or two blocks of the partial snapshots, Superblock FTL takes a full snapshot. Writing a partial snapshot results in 1.56% of extra page write operations when a block consists of 64 pages. As the utilization

[2]The PGD size is 256KB. The size of valid page counters is 1,792KB, and the list of U-blocks and free blocks occupies 192KB.

of a block decreases, the overhead tends to grow. However, since the average utilization in Superblock FTL is usually higher than 90%, the added overhead remains less than 2% of total write operations.

The location of both the full snapshot and partial snapshot are indirected from the first few fixed blocks of flash memory. If a full snapshot is taken or blocks are allocated for partial snapshot, a page keeping the locations is written to the fixed areas. To prevent wear out of the fixed blocks, we can use multiple levels of indirections. Note that this technique has been popular in flash-based storage systems [Yim et al. 2005; Bityutskiy 2005].

During the system start-up, Superblock FTL rebuilds the up-to-date data structures from snapshots. It first initializes in-memory data structures with the information in the last full snapshot. Then Superblock FTL modifies data structures reflecting the information stored in any partial snapshots taken after the full snapshot. Finally, Superblock FTL scans those pages in U-blocks written after the last partial snapshot, updating PGD, PMDs, and PTs (or map pages) with the information stored in spare areas (DI), accordingly. For 32GB SLC NAND flash memory, the full snapshot size is about 984KB. If two blocks are assigned for logging partial snapshots, there are at most 128 partial snapshots to be read. With the partial snapshots, Superblock FTL replays allocation and page writes by modifying PGDs. After the last partial snapshot, the locations of updated pages are unknown. In order to find these pages, we should investage all U-blocks and scan the pages updated after the last partial snapshot. This requires at least two spare reads for each U-block for identifying its logical block address and the end of the log in it. In addition, one spare read for each written page. For 32GB SLC NAND, 8,192 extra blocks are assigned. Therefore, about 16,640 spare areas including those of updated pages should be scanned. Overall, the estimated initialization time is about 587.9ms. If 32GB MLC NAND flash memory (with the map page) is employed and one block is assigned to partial snapshots, the full snapshot size is roughly 70KB and at most 128 partial snapshots exist. For 32GB MLC NAND, we currently allocate 2,048 extra blocks; hence, at least 4,096 spare areas need to be scanned. As a result, the initialization time is estimated as about 283.0ms.

Wear leveling is also an important issue in designing flash-based storage, since NAND flash memory has limited erase cycles. However, wear leveling is beyond the scope of this article, being a subject of another study. Other researchers already proposed several wear leveling schemes that can be applied to numerous FTLs [Jung et al. 2007; Chang 2007]. Furthermore, an abstraction layer such as UBI [(MTD) 2008] has been proposed that provides wear leveling transparent to the upper layer. Therefore, Superblock and other FTLs can adopt one of these solutions to achieve wear leveling.

## 5. PERFORMANCE EVALUATION

This section evaluates the performance of the proposed Superblock FTL. For comparison, we have also evaluated two previous block-mapped FTL schemes, the log block scheme [Kim et al. 2002] and FAST [Lee et al. 2007], and one page-mapped FTL scheme, DAC (Dynamic dAta Clustering) [Chiang et al. 1999].

Table IV. Traces used for Evaluation

| Trace | Description | Total Storage Size (MB) | The Number of Sectors Written |
|---|---|---|---|
| PIC | This trace models the workload of digital cameras. Picture files whose average size is 1.9MB are created and deleted. | 8,192 | 55,145,707 |
| MP3 | This trace models the workload of MP3 players. MP3 files whose average size is 4.4MB are created and deleted. | 8,192 | 56,237,750 |
| MOV | This trace models the workload of movie players. Movie files whose average size is 681MB are created and deleted. | 8,192 | 54,360,116 |
| PMP | This trace models the workload of portable media players (PMPs). A number of picture files, MP3 files, and movie files are created and deleted. | 8,192 | 55,614,913 |
| PC | This trace is extracted from a real user activity on a desktop of personal usage during 5 days. | 32,768 | 28,951,277 |
| PCMark | This trace is obtained by running PCMark05 HDD tests five times. The tests consist of Windows XP start-up, general application loading, general hard disk usage, virus scanning, and writing files. | 32,768 | 19,477,495 |
| Install | This trace is collected during the installation of general applications in SYSmark 2007 preview. | 40,960 | 24,472,089 |
| SYSmark | This trace is gathered while performing all scenarios in SYSmark 2007 preview. The scenarios include e-learning, office productivity, video creation, and 3D modeling with real applications. | 40,960 | 32,962,880 |

## 5.1 Evaluation Methodology

We have implemented trace-driven simulators for the log block scheme, FAST, DAC, and the proposed Superblock FTL. The workload is chosen to reflect the representative storage access patterns of multimedia mobile devices and laptop computers. Table IV summarizes the characteristics of traces used in this article. These traces are extracted from disk access logs on FAT32 and NTFS file systems by using DiskMon [Russinovich 2006]. Four traces, PIC, MP3, MOV, and PMP, synthetically model the workload of digital cameras, MP3 players, video players, and portable media players. The PC trace is the storage access trace of a real user during 5 days, which includes Web surfing, e-mailing, word processing, preparing presentations, and playing MP3 songs and movies.

PCMark [Futuremark Corp. 2005] and SYSmark [BAPCo 2007] are popular benchmarks for desktop or laptop computers. PCMark is a series of synthetic benchmark tools that measure the overall performance and the performance of individual components, such as CPU, memory, graphics card, and hard disk drive (HDD), in typical home PC usage. Our PCMark trace is obtained from HDD Test suite, which is one of PCMark test suites where the performance of the hard disk is measured by simulating Windows XP start-up, application loading, general hard disk usage for several common applications, virus scanning, and file writes. SYSmark is an application-based benchmark that reflects usage patterns of business uses in the areas of video creation, e-learning, 3D modeling, and office productivity. SYSmark differs from PCMark in that SYSmark emulates the real-world scenarios by actually installing popular applications that many people use every day such as Microsoft Office, Adobe Photoshop, WinZip, and many others. The Install trace represents disk activities during the installation of these applications in SYSmark, and the SYSmark trace is obtained while the SYSmark benchmark is running. The first four traces are from systems running on FAT32 file system and model the usage pattern for mobile embedded devices, while the others are for general computer systems running on NTFS file system.[3]

The main performance metric we use is the total merge cost $MergeCost(\mathbf{W})$ for a given trace $\mathbf{W}$. As described in Section 3.1, it is a function of the number of erase operations and the number of valid pages copied during merge operations. The simulators model the timing parameters related to current technologies as exactly as possible. The actual value of $MergeCost(\mathbf{W})$ is calculated based on the parameters of the large block SLC and MLC NAND flash memory shown in Table I. In addition, we have measured the average utilization of U-blocks, $U_{avg}$, for each FTL scheme. The utilization of a U-block is defined as a fraction of written pages when the block is selected as a victim during the merge operation. The evaluated NAND flash memory size ranges from 8GB to 40GB depending on the configuration where each trace is obtained (see Table IV).

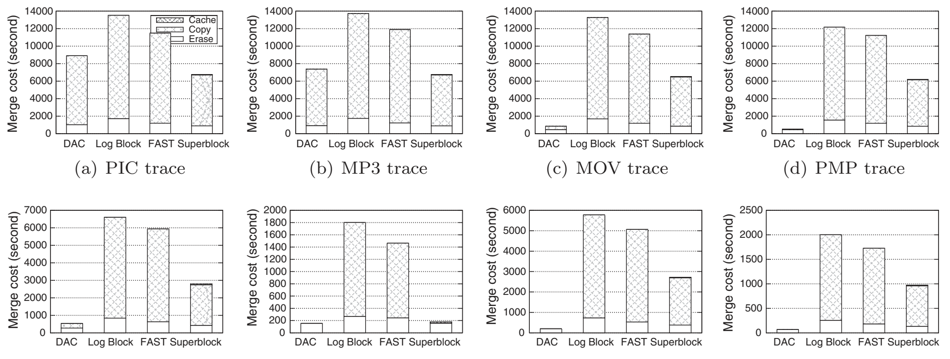
Unless otherwise stated explicitly, we have performed our experiments in the following conditions. The superblock size is four ($N = 4$), and the number of map cache entries is 16.[4] The number of available U-blocks is initially configured as 2,048 for 8GB, and 8,192 for 32GB and 40GB. These numbers correspond to 2.5% to 3.1% of the total number of blocks in NAND flash memory.[5] Before each run starts, the whole D-blocks are filled with valid pages so there are no free pages available in D-blocks. Our experiments are performed both on the large block SLC NAND and on the MLC NAND flash memory, but due to space limitation, our analysis focuses on the results obtained on the large block SLC NAND flash memory. Results on the MLC NAND flash are briefly given in Section 5.8.

## 5.2 Overall Performance for Original Traces

Figure 13 compares the merge cost, $MergeCost(\mathbf{W})$, for each FTL scheme. There are four bars for each trace, which correspond to the result of DAC, the log block scheme, FAST, and Superblock FTL, respectively. We break down the merge cost into the time spent on manipulating map cache entries (CACHE), copying valid pages (COPY), and performing erase operations (ERASE).

First, we can observe that DAC outperforms other block-mapped FTLs in most traces except for PIC and MP3 traces. This is because the flexibility of

[3]The results of the last four traces (PC, PCMark, Install, and SYSmark) are included in this article as the proposed Superblock FTL can be also used inside flash SSDs (solid state disks).
[4]One entry can cache the mapping information stored in a spare area of a single page.
[5]It is known that most commercial NAND flash-based storage reserves about 5% of the total number of blocks for the purpose of FTL's internal use such as U-blocks, bad block remapping, and snapshots.



(a) PIC trace (b) MP3 trace (c) MOV trace (d) PMP trace

(e) PC trace (f) PCMark trace (g) Install trace (h) SYSmark trace

Fig. 13. Comparisons of the merge cost in DAC, the log block scheme, FAST, and Superblock FTL for original traces.

the page-mapped FTL scheme effectively avoids the overhead in performing full merge operations as much as possible by clustering pages with similar update frequency into the same region. In PIC and MP3 traces, it turns out that the number of U-blocks is not enough to identify hot pages from cold pages accurately. On the other hand, the copy costs of DAC over MOV and PMP traces are very small, since the average file size is big enough to correctly cluster pages with 2,048 U-blocks. Our analysis indicates that the overhead of DAC is heavily influenced by the number of available U-blocks. In Section 5.5, we will examine the impact of the number of U-blocks on the merge cost in more detail.

Overall, Superblock FTL exhibits noticeably smaller merge costs than other block-mapped FTL schemes. Superblock FTL outperforms FAST by reducing the merge cost by 41% to 88% over the whole traces. In particular, most of the benefits come from the decrease in the number of valid pages copied during merge operations; in comparison to FAST, Superblock FTL reduces the time spent on copying valid pages by up to 99.8% for PCMark, and by 43% to 56% for other traces. The result of Superblock FTL over PCMark trace is outstanding as comparable as that of DAC. Since the number of blocks touched by the PCMark trace is small, every superblock can have four U-blocks (the maximum value), and merge operations are delayed as long as possible.

We also point out that the map cache manipulation time in Superblock FTL is almost negligible. This includes the time to read the page-mapping information from spare areas as necessary, but the overhead is only 0.8% to 1.7% of the total merge cost over all traces except for PCMark. In the PCMark trace, the cache manipulation time occupies 11.8% of the merge cost (*MergeCost*(**W**)), but the time accounts for only 0.9% of the total write cost (*WriteCost*(**W**)).

## 5.3 Overall Performance for Aligned Traces

Apparently, the performance of block-mapped FTLs will increase if a large amount of data are written sequentially, since it increases the chance of switch merge operations; U-blocks are sequentially updated and they can be switched

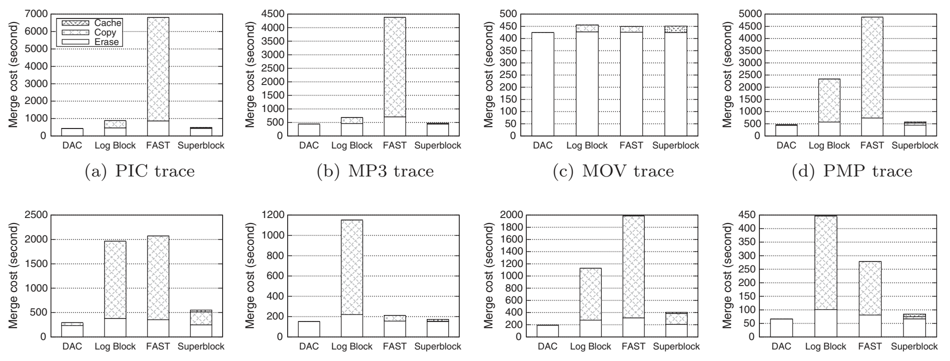
Table V. The Proportion of Aligned and Unaligned Requests for 2KB Page Size

| | Original Trace | | Aligned Trace | |
|---|---|---|---|---|
| Trace | Aligned | Not Aligned | Aligned | Not Aligned |
| PIC | 0.1% | 99.9% | 98.2% | 1.8% |
| MP3 | 1.1% | 98.9% | 98.8% | 1.2% |
| MOV | 0.2% | 99.8% | 99.8% | 0.2% |
| PMP | 0.7% | 99.3% | 99.2% | 0.8% |
| PC | 1.7% | 98.3% | 93.4% | 6.6% |
| PCMark | 1.3% | 98.7% | 97.3% | 2.3% |
| Install | 0.6% | 99.4% | 97.9% | 2.1% |
| SYSmark | 0.7% | 99.3% | 97.4% | 2.6% |

with the corresponding D-blocks without copying valid pages. In Figure 13, however, we can see that block-mapped FTLs consume most of their time on copying valid pages even for PIC, MP3, MOV, and PMP traces, which are expected to show relatively sequential write patterns. Specifically, the number of copied pages exceeds the number of pages written in these traces and, unlike our expectations, full merge operations are mostly performed. This implies that most write requests are not handled as sequential ones.

We have investigated the reason of this phenomenon, and it is revealed that most write requests are not aligned to the page size of NAND flash memory. In consequence, the last page of a request is overlapped with the first page of the following sequential request. For instance, assume that there are two sequential write requests $w_1$ and $w_2$, where $w_1$ writes 8 sectors from LBA #3 to #10 and $w_2$ writes another 8 sectors thereafter from LBA #11 to #18. When the page size is 2KB, $w_1$ is essentially treated by FTL as a write request of 3 pages from LPA (logical page address) #0 to #2, and $w_2$ as a write request of 3 pages from LPA #2 to #4. Even if the original requests form a sequential write pattern, FTL regards this as nonsequential write requests where the overlapped page (LPA #2) is updated twice. When $w_1$ is arrived, the overlapped page is first written to a U-block, and then the page is rewritten to the same U-block while $w_2$ is processed. As a result, it will be required to perform an expensive full merge operation later to reclaim the U-block, instead of a switch merge operation.

The left column in Table V shows the proportion of aligned and unaligned requests in each trace when the page size is 2KB. We can see that the starting LBAs of almost every write request are not aligned to the page boundary. We believe the generation of such unaligned write requests is closely related to the location of the disk partition where the trace is collected. In general, a disk partition can start at any point of a hard disk, and our study shows that the first disk partition usually begins at the 63rd sector. Since the unalignment is originated from the initial location of the particular disk partition, we have adjusted our traces by shifting sector numbers by a fixed offset. The right column in Table V presents the resulting statistics on the proportion of aligned and unaligned requests. We can observe that most requests are now aligned to 2KB page size.



(a) PIC trace (b) MP3 trace (c) MOV trace (d) PMP trace

(e) PC trace (f) PCMark trace (g) Install trace (h) SYSmark trace

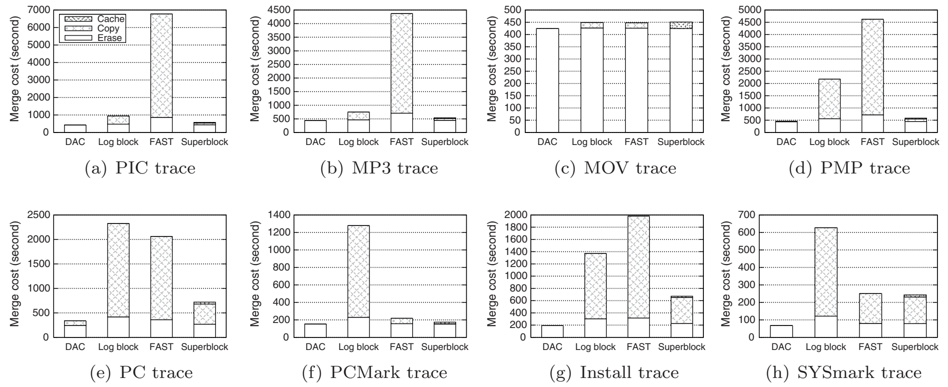
Fig. 14. Comparisons of the merge cost in DAC, the log block scheme, FAST, and Superblock FTL for aligned traces.

Figure 14 depicts the merge costs simulated with the aligned traces. For all traces, the merge costs of the log block scheme, FAST, and Superblock FTL are significantly reduced, especially in the copy overhead. Since a large portion of disk accesses are now sequential, U-blocks can be fully written, eventually being switched with the corresponding D-blocks.

As shown in Figure 14, Superblock FTL still outperforms other block-mapped FTLs, and in many cases its performance is comparable to DAC. Over PIC, MP3, and PMP traces where most requests are sequential and a large number of files are created and deleted, the merge cost of Superblock FTL is lower than the log block scheme and FAST by 30% to 76% and 88% to 93%, respectively. In PC, Install, and SYSmark traces, the trend is similar; Superblock FTL reduces the merge cost by 70% to 80% compared to FAST, mainly in the copy cost. For PCMark trace, Superblock FTL outperforms FAST by 19%. All the evaluated FTLs show very similar performance in MOV, as the trace is highly sequential with only a small fraction of metadata updates. Overall, we can see that Superblock FTL is indeed quite effective in reducing the overall merge cost, while imposing very little management overhead.

Comparing Figure 14 with Figure 13, we observe that FAST shows a relatively small performance gain with aligned traces. This is because FAST frequently misinterprets directory entry updates as sequential accesses. If a request is about to update the first page of a block, FAST considers it as the start of sequential writes and allocates a sequential log block. When the prediction fails, however, this strategy may come at high cost, involving full merge operations for sequential log blocks.

In recent high-performance flash storage devices such as SSDs, manufacturers often employ a small memory buffer to mitigate the problem associated with the trace alignment. To investigate the impact of the buffer, we have added an 1MB write buffer to our simulator and examined changes in the write performance with unaligned traces. As depicted in Figure 15, the merge cost has been significantly reduced in all traces. Over PIC, MP3, MOV, and PMP traces, the merge costs of both the log block scheme and Superblock FTL are



(a) PIC trace (b) MP3 trace (c) MOV trace (d) PMP trace

(e) PC trace (f) PCMark trace (g) Install trace (h) SYSmark trace

Fig. 15. Comparisons of the merge cost in DAC, the log block scheme, FAST, and Superblock FTL with an 1MB write buffer.

reduced by 82.1% to 96.6% compared to the results with unaligned traces shown in Figure 13. DAC exhibits almost the same merge costs compared to those measured with aligned traces. In case of FAST, the merge costs are lowered by 41.0% to 96.1%, with deviating from the results with aligned traces by at most 5.2%.

The results evaluated with PC, PCMark, Install, and SYSmark traces also show the similar trend, reducing the merge cost by up to 85.5% compared to Figure 13. These traces exhibit slightly smaller improvement than PIC, MP3, MOV, and PMP traces. This is because these traces contain a large number of small random accesses.

Our evaluation results show that the use of a small write buffer is quite effective in improving the write performance especially when the requests from file systems are not aligned to the page boundary. Since the merge cost easily varies by more than a factor of 10 depending on the existence of unaligned write requests, a more fundamental approach is to let file systems manage disk blocks whose offsets are aligned to the page size. Recently, the International Disk Drive Equipment and Materials Association (IDEMA) [2007] has released the Long Data Block standards. The objective of the standards is to use a new 4KB sector size, instead of the traditional 512-byte sector size. Hence, we expect that every disk access request will be aligned to 4KB boundary in the near future. As aligned traces reflect the actual behavior of storage access pattern more accurately and ease the analysis, we only show the results of aligned traces in the following discussions.

## 5.4 The Detailed Analysis of the Merge Costs

The detailed analysis of the merge cost under the aligned PC trace is presented in Table VI including previous FTL schemes and Superblock FTLs with several different configurations. In Table VI, the superblock size of SB-1-IP and SB-1-OOP is one, while that of SB-4-IP and SB-4-OOP is four. D-blocks are arranged by the in-place scheme in SB-1-IP and SB-4-IP and by the out-of-place scheme in SB-1-OOP and SB-4-OOP. Table VI compares $U_{avg}$, $MergeCount(\mathbf{W})$, and $MergeCost_{avg}(\mathbf{W})$.

Table VI. The Detailed Analysis of the Merge Cost for (Aligned) PC Trace in DAC, the Log Block Scheme, FAST, and Superblock FTL with Several Different Configurations

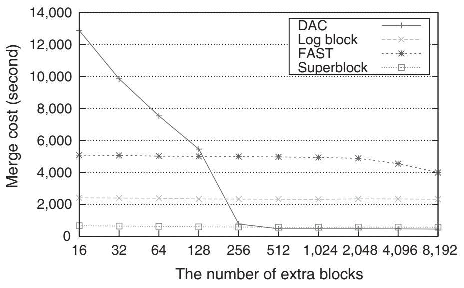
| | DAC | Log block | FAST | SB-1-IP | SB-1-OOP | SB-4-IP | SB-4-OOP |
|---|---|---|---|---|---|---|---|
| $U_{avg}$ | 100.0% | 84.8% | 74.2% | 89.7% | 90.5% | 96.0% | 95.4% |
| *MergeCount* (**W**) | 38,045 | 134,561 | 153,791 | 127,300 | 127,236 | 118,910 | 110,137 |
| $MergeCost_{avg}$(**W**) (millisecond) | 7.7 | 14.6 | 13.5 | 8.4 | 6.2 | 10.2 | 5.0 |
| *MergeCost* (**W**) (second) | 293.0 | 1,963.7 | 2,074.4 | 1,070.0 | 793.5 | 1,215.7 | 550.5 |

SB-$n$-$p$ represents a Superblock FTL scheme with the superblock size $n$ and the placement scheme $p$. The placement scheme $p$ is either IP or OOP, which denotes the in-place scheme or the out-of-place scheme, respectively. SB-4-OOP also tries to separate hot pages from cold pages during the merge operation and corresponds to the scheme labeled "*Superblock*" in Figure 14.

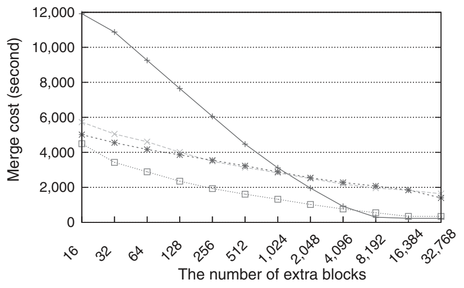
In FAST, *MergeCount* (**W**) is increased by 14.3% compared to the log block scheme due to the decrease in $U_{avg}$. This is because FAST incorrectly classifies nonsequential write requests into sequential ones. Accordingly, sequential log blocks are frequently merged before they are fully used. $MergeCost_{avg}$(**W**) is decreased by 7.5%, as most of the increased number of merge operations are partial merges for sequential log blocks. According to our simulation results, the number of full merge operations in the log block scheme and FAST is 54,976 and 38,659, respectively, while the number of partial merge operations is increased from 4,316 (the log block scheme) to 52,127 (FAST). Even for other traces such as PIC, MP3, and PMP, FAST shows $U_{avg}$ value between 73.5% to 82.3%, which is far less than that of the log block scheme ranging from 98.5% to 99.5%. The original motivation of FAST is to share U-blocks among all the D-blocks to improve $U_{avg}$, but it does not work out that way if it fails to isolate sequential write requests properly. On the other hand, FAST shows better performance than the log block scheme over PCMark and SYSmark traces, as there are not many sequential requests in these traces and the performance of FAST is not interfered with them.

The difference between the log block scheme and SB-1-IP is that SB-1-IP allocates more than one U-block for a logical block in order to exploit the block-level temporal locality, as mentioned in Section 3.2. Both *MergeCount* (**W**) and $MergeCost_{avg}$(**W**) of SB-1-IP have been reduced by 5.4% and 42.5%, respectively, compared to the log block scheme. Since the pages in a block can be distributed over numerous U-blocks in SB-1-IP, the merge operation can be delayed much longer. Moreover, a single merge operation in SB-1-IP usually produces many free blocks at once.

We have further opportunity to cut down $MergeCost_{avg}$(**W**) by managing D-blocks with the out-of-place scheme as the result of SB-1-OOP indicates. The in-place scheme requires extra full merge operations to meet the programming restriction of the large block SLC or MLC NAND flash memory, as described in Section 2.3. Using the out-of-place scheme not only eliminates such extra full merge operations but also increases the chance of partial merge operations. As a result, when we move from SB-1-IP to SB-1-OOP, $MergeCost_{avg}$(**W**) is decreased by 26.2%.



(a) Merge cost over the aligned PMP trace.



(b) Merge cost over the aligned PC trace.

Fig. 16. The impact of the number of U-blocks on the merge cost (aligned traces).

When we increase the superblock size from one (SB-1-OOP) to four (SB-4-OOP), $U_{avg}$ is raised from 90.5% to 95.4% due to the block-level spatial locality. Accordingly, *MergeCount* (**W**) is reduced by 13.4%. Note that SB-4-OOP also implements the merge algorithm described in Section 4.3, which separates hot pages from cold pages within a superblock. We can confirm that such strategy is effective in suppressing the growth in $MergeCost_{avg}(\mathbf{W})$ despite of the increased degree of sharing.

## 5.5 The Effect of the Number of U-Blocks

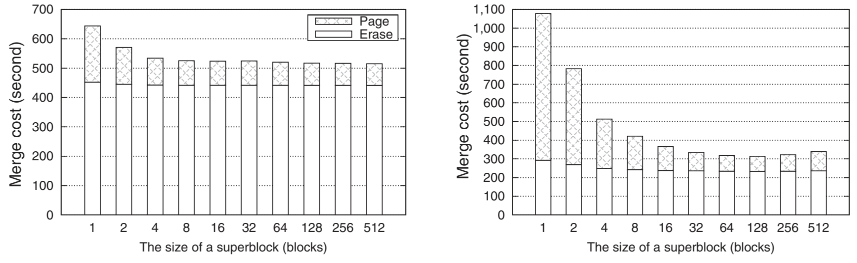
Figure 16 illustrates the changes in the merge cost over two traces, PMP and PC, with respect to the total number of U-blocks. Again, each graph corresponds to the result of DAC, the log block scheme, FAST, and Superblock FTL, respectively. As the amount of U-blocks is raised, the merge cost gradually falls in all schemes. This is an expected result, since merge operations are delayed if there are more free blocks. In particular, we can observe that the performance of DAC is quite sensitive to the number of free blocks, which is in line with previous findings [Kawaguchi et al. 1995; Chiang et al. 1999; Kim et al. 2002; Chang and Kuo 2005]. Block-mapped FTLs are less affected by the number of U-blocks, especially when most of write requests are sequential as can be seen in the PMP trace (see Figure 16(a)).

Figure 16 shows that Superblock FTL defeats other block-mapped FTL schemes for the same number of U-blocks. Superblock FTL even outperforms DAC when the number of available U-blocks is relatively small.

## 5.6 The Effect of the Superblock Size

Figure 17 investigates the impact of the superblock size on the merge cost for aligned PMP and PC traces. For this experiment, we held all the page-mapping information in RAM without storing them in spare areas, since the current implementation does not support the superblock size greater than eight.

As the superblock size grows, the merge cost is decreased because the storage utilization of U-blocks increases due to the block-level spatial locality. When the superblock size is increased from one to four, the merge costs over PMP and PC traces are reduced by 17.0% and 52.4%, respectively. The PC trace is more sensitive to the superblock size (see Figure 17(b)), with the merge cost improved



(a) Merge cost over the aligned PMP trace. (b) Merge cost over the aligned PC trace.

Fig. 17. The impact of the superblock size on the merge cost (aligned traces).

by up to 70.8% at the superblock size of 128. On the contrary, the multimedia workload is less affected by the superblock size, as shown in Figure 17(a).

If the superblock size goes beyond some point, the merge cost begins to remain stable or even increase. This is because the larger degree of sharing in U-blocks tends to increase $\mathit{MergeCost}_{avg}(\mathbf{W})$, while the benefit from the higher storage utilization diminishes as the superblock size grows. In particular, the cost of full merge operations grows as larger superblock size is used. According to our simulation results, $\mathit{MergeCost}_{avg}(\mathbf{W})$ is increased by 9.4% (from 3.2ms to 3.5ms) when the superblock size is increased from 128 to 512 for the PC trace. Meanwhile, $\mathit{MergeCount}(\mathbf{W})$ is decreased only by 1.3%.

## 5.7 The Effect of the Cache Size

Figure 18(a) presents the changes in the map cache hit ratios with respect to the number of cache entries for the PC trace. The cache hit ratio is calculated as follows.

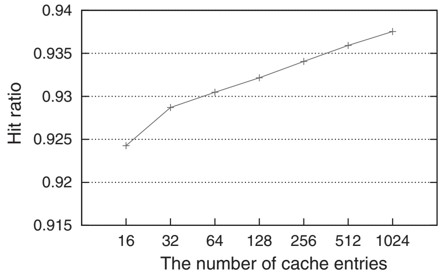
$$\text{Hit ratio} = \frac{\text{Map Cache Reads}}{\text{PMD/PT Reads}} \tag{4}$$

Whenever Superblock FTL reads or writes a page, it requires one or two spare accesses. If PMD and PT is stored in the same spare area, only one spare access is enough. Otherwise, one for PMD entry and the other for PT entry are necessitated.

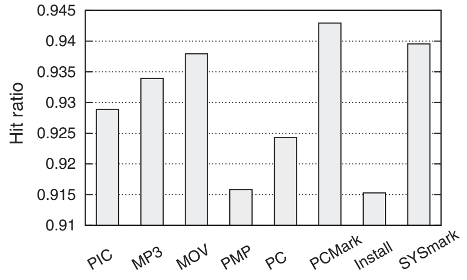
From the results, we can notice that the cache hit ratio is slightly improved from 92.4% to 93.8% as the number of cache entries increases from 16 to 1,024. The improvement is only 1.4% at most, as illustrated in Figure 18(a). Thus, 16 cache entries seem to be sufficient in most cases.

Figure 18(b) illustrates hit ratios in the map cache across all the aligned traces with 16 cache entries. For all tested workloads, the hit ratio is equal to or greater than 91.5%. Due to the high hit ratio in the map cache, the cache-management overhead hardly affects the overall performance, as shown in Figure 14. The main reason of these high hit ratios is that most requests consist of multiple pages and numerous requests are sequential so that the same PMD and PT entries are accessed in a row.

In Section 4.2.1, we briefly mentioned that it is possible to construct the page-level mapping information without using the proposed hybrid mapping



(a) The cache hit ratio according to the number of cache entries (aligned PC trace).



(b) The cache hit ratios across all traces (aligned).

Fig. 18. Hit ratios for map cache.

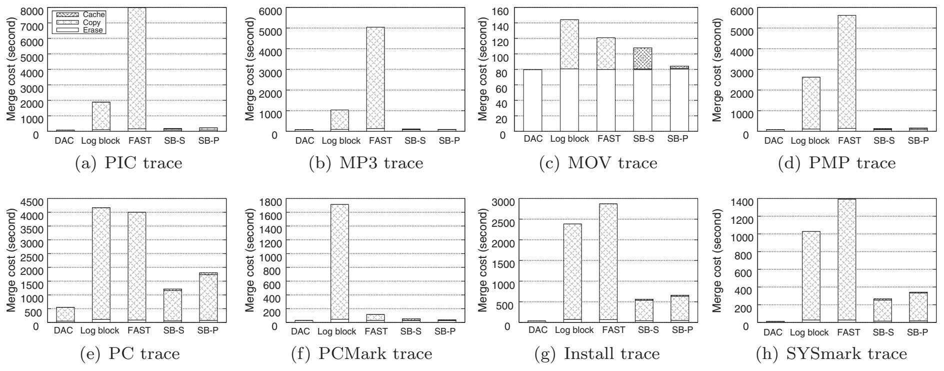


Fig. 19. Comparisons of the merge cost on MLC NAND flash memory (aligned traces).

scheme by scanning all the spare areas in a superblock on every cache miss. Even though the hit ratio is pretty high, this strategy is not practical because the total number of page accesses is significant and the miss penalty is also considerable. Scanning spare areas of a superblock, which we assume consists of four blocks, requires about 7.8ms, and it takes even longer if additional U-blocks are assigned to the superblock. The total number of page write requests in the PC trace is over 11 million excluding those generated during garbage collection. In this case, 8.5% of cache miss ratio results in at least 7,293.0 seconds of the additional cache access cost, which increase the total overhead by 1,387.2%.

## 5.8 The Results on MLC NAND Flash Memory

Figure 19 depicts the results simulated on MLC NAND flash memory. In the figure, SB-S denotes the original Superblock FTL, which keeps the page-level mapping information in spare areas. On the contrary, SB-P represents Superblock FTL which utilizes map pages for the same purpose. In SB-P, the map cache size is set to four entries for PIC, MP3, MOV, and PMP traces, and 16 entries for other traces. Although the map cache size is configured to 4 or 16 pages, the overall mapping information size of SB-P is smaller than that of SB-S due to the reduced size of PGD. For fair comparison, we have also reduced the number of extra blocks in SB-P by the amount of space reserved for map pages.

Similar to previous results on SLC NAND flash memory, Superblock FTL notably outperforms other block-mapped FTLs. In many cases, Superblock FTL shows comparable performance to DAC. Several factors affect the merge cost on MLC NAND either positively or negatively. The positive side is that since the block size is quadrupled, a single erase operation generates more free space. In addition, when there is enough block-level temporal and spatial locality, a single U-block can absorb larger number of update requests. This will reduces the number of merge operations. However, there are several negative sides that adversely affect the FTL performance. First, since the page size is doubled, reading or updating a small area inside a page takes relatively more time. Second, the write latency of MLC NAND flash memory is much longer than that of SLC NAND. The quadrupled block size also increases the number of valid pages in a given block. Therefore, the cost of copying valid pages becomes more expensive. Third, if many blocks are updated rather randomly, the utilization of U-blocks will drop and the increased block size can introduce considerable merge overhead. Finally, in Superblock FTL using map pages, the chance of switch merge and partial merge operations is lowered due to map page update.

For multimedia traces such as PIC, MP3, MOV, and PMP, moving toward MLC NAND from SLC NAND is beneficial due to the increased block size, lowering the overall merge cost of Superblock FTL by a factor of 2.6 to 4.2 when the page-level mapping information is stored in spare areas (SB-S). In the results of Superblock FTL, the cost of erase is significantly reduced, while the overhead of copying pages is slightly augmented. In PC, Install, and SYSmark traces, however, the performance of Superblock FTL is degraded by a factor of 1.4 to 3.2 by switching from SLC to MLC. The reason is that these traces exhibit more random access patterns and MLC NAND adds more cost in copying valid pages. Although the merge count is reduced by a factor of 3.6 to 4.0 in the PC, Install, and SYSmark traces, the average merge cost has increased more rapidly by 5.3 to 11.5 times.

If the page-level mapping information is kept in map pages, Superblock FTL exhibits slightly higher erase and copy costs in most traces. Over multimedia traces, SB-P shows 1.0 to 40.2% of additional erase and copy costs compared to SB-S. This is because map page updates interfere with the possibility of switch and partial merge operations. If a map page is written in a block more than twice, the block should be fully merged later. On the other hand, the large cache space and the wider coverage of map pages lower the cache access overhead by 81.9 to 89.7%. In case of PC, Install, and SYSmark traces, the increment in the overhead is 18.3 to 49.9% due to more frequent map page updates. As a result, the ratio of full merge operation is increased. Similar to the results over multimedia traces, the cost of accessing map cache is lowered by 1.6 to 2.4% except for the PC trace. The PC trace suffers from 20.4% of the additional map cache cost, since the request pattern shows weaker locality than the other traces.

Table VII. The Detailed Statistics of Merge Operations on MLC NAND Flash Memory

| | DAC | Log Block | FAST | SB-S | SB-P | DAC | Log Block | FAST | SB-S | SB-P |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | **PIC** | | | | | **MP3** | | |
| Full merge (%) | 17,991 (100) | 12,665 (22.7) | 21,788 (28.8) | 406 (0.8) | 865 (1.6) | 18,329 (100) | 6,795 (12.2) | 13,570 (19.8) | 75 (0.1) | 82 (0.1) |
| Partial merge (%) | 0 (0) | 631 (1.1) | 37,918 (50.0) | 403 (0.7) | 205 (0.4) | 0 (0) | 153 (0.3) | 24,331 (35.5) | 0 (0) | 0 (0) |
| Switch merge (%) | 0 (0) | 42,558 (76.2) | 16,043 (21.2) | 53,032 (98.5) | 53,293 (98.0) | 0 (0) | 48,807 (87.5) | 30,669 (44.7) | 54,810 (99.9) | 55,223 (99.9) |
| *MergeCount* | 17,991 | 55,854 | 75,749 | 53,841 | 54,363 | 18,239 | 55,755 | 68,570 | 54,885 | 55,305 |
| $MergeCost_{avg}$ | 4.5 | 33.7 | 105.4 | 3.5 | 4.2 | 4.5 | 18.6 | 73.5 | 2.2 | 1.7 |
| *MergeCost* | 81.4 | 1883.4 | 7986.5 | 189.9 | 227.7 | 82.5 | 1036.6 | 5039.0 | 120.0 | 94.7 |
| $U_{avg}$ (%) | 100 | 96.6 | 71.3 | 98.0 | 97.7 | 100 | 98.6 | 80.2 | 100 | 100 |
| | | | **MOV** | | | | | **PMP** | | |
| Full merge (%) | 17,698 (100) | 417 (0.8) | 220 (0.4) | 0 (0) | 16 (0) | 18,279 (100) | 18,140 (32.0) | 15,924 (22.5) | 164 (0.3) | 417 (0.8) |
| Partial merge (%) | 0 (0) | 77 (0.1) | 429 (0.8) | 0 (0) | 0 (0) | 0 (0) | 268 (0.5) | 24,493 (34.6) | 127 (0.2) | 45 (0.1) |
| Switch merge (%) | 0 (0) | 52,829 (99.1) | 52,648 (98.8) | 53,063 (100) | 53,472 (100) | 0 (0) | 38,333 (67.5) | 30,346 (42.9) | 54,456 (99.5) | 54,781 (99.1) |
| *MergeCount* | 17,698 | 53,323 | 53,297 | 53,063 | 53,488 | 18,279 | 56,741 | 70,763 | 54,747 | 55,243 |
| $MergeCost_{avg}$ | 4.5 | 2.7 | 2.3 | 2.0 | 1.6 | 4.5 | 46.2 | 79.4 | 2.5 | 2.8 |
| *MergeCost* | 79.6 | 144.1 | 120.8 | 107.8 | 84.1 | 82.3 | 2,619.9 | 5,616.9 | 136.4 | 152.2 |
| $U_{avg}$ (%) | 100 | 99.6 | 99.4 | 100 | 100 | 100 | 96.6 | 77.5 | 99.4 | 99.1 |
| | | | **PC** | | | | | **PCMark** | | |
| Full merge (%) | 9,596 (100) | 28,733 (61.1) | 12,335 (28.2) | 4,700 (15.8) | 8,780 (26.8) | 6,353 (100) | 12,153 (63.8) | 468 (2.4) | 18 (0.1) | 15 (0.1) |
| Partial merge (%) | 0 (0) | 968 (2.1) | 16,972 (38.7) | 2034 (6.9) | 975 (3.0) | 0 (0) | 0 (0) | 910 (4.7) | 0 (0) | 0 (0) |
| Switch merge (%) | 0 (0) | 17,318 (36.8) | 14,501 (33.1) | 22,955 (77.3) | 22,949 (70.2) | 0 (0) | 6,900 (36.2) | 17,975 (92.9) | 18,990 (99.9) | 19,182 (99.9) |
| *MergeCount* | 9,596 | 47,019 | 43,808 | 29,689 | 32,704 | 6,353 | 19,053 | 19,353 | 19,008 | 19,197 |
| $MergeCost_{avg}$ | 57.1 | 88.6 | 91.4 | 53.0 | 55.2 | 4.5 | 89.9 | 6.0 | 2.8 | 2.0 |
| *MergeCost* | 547.8 | 4,165.4 | 4,003.0 | 1,215.6 | 1,805.0 | 28.6 | 1,713.5 | 115.8 | 53.0 | 37.9 |
| $U_{avg}$ (%) | 100 | 61.3 | 65.8 | 86.1 | 80.0 | 100 | 100 | 96.7 | 100 | 100 |
| | | | **Install** | | | | | **SYSmark** | | |
| Full merge (%) | 8,021 (100) | 16,398 (49.8) | 8,881 (27.0) | 1,754 (7.1) | 2,724 (10.7) | 2,778 (100) | 7,066 (62.4) | 3,760 (31.0) | 736 (8.2) | 1,556 (16.8) |
| Partial merge (%) | 0 (0) | 695 (2.1) | 12,510 (38.0) | 736 (2.9) | 233 (0.9) | 0 (0) | 323 (2.8) | 5,767 (47.6) | 702 (7.8) | 245 (2.6) |
| Switch merge (%) | 0 (0) | 15,807 (48.1) | 11,496 (35.0) | 22,336 (90.0) | 22,453 (88.4) | 0 (0) | 3,939 (34.8) | 2,588 (21.4) | 7,568 (84.0) | 7,463 (80.6) |
| *MergeCount* | 8,021 | 32,900 | 32,887 | 24,826 | 25,410 | 2,778 | 11,328 | 12,115 | 9,006 | 9,264 |
| $MergeCost_{avg}$ | 4.5 | 72.5 | 87.3 | 22.6 | 25.9 | 4.5 | 90.7 | 115.0 | 29.6 | 36.9 |
| *MergeCost* | 36.2 | 2,384.0 | 2,870.2 | 560.5 | 657.8 | 12.5 | 1,027.5 | 1,392.8 | 266.2 | 342.2 |
| $U_{avg}$ (%) | 100 | 73.1 | 73.2 | 92.4 | 90.9 | 100 | 73.6 | 68.8 | 89.6 | 87.5 |

*MergeCost* and $MergeCost_{avg}$ are given in seconds and in milliseconds, respectively.

Table VII compares *MergeCount* (**W**), $MergeCost_{avg}$(**W**), *MergeCost* (**W**), and $U_{avg}$ for each trace in detail. Specifically, the number of merge operations are further classified according to the type of each merge operation: full, partial, or switch merge. From Table VII, we notice that Superblock FTL is successful in reducing the number of merge operations, as several logical blocks in a superblock share a U-block with effectively increasing the utilization of U-blocks ($U_{avg}$). At the same time, the average merge cost $\mathit{MergeCost}_{avg}(\mathbf{W})$ has been also improved compared to other block-mapped FTLs. This is because the ratio of switch merge operations is significantly increased by (i) using fine-grain address translation inside a superblock and (ii) separating hot pages from cold pages during merge operations. On the contrary, when a logical block is not sequentially written from the beginning, the full merge operation is inevitable in the log block scheme and FAST.

## 5.9 Memory Consumption of Mapping Information

In Superblock FTL, the mapping information size is given by the summation of the PGD size and the map cache size. The PGD size depends on the flash memory size, while the map cache size is configurable as needed. Assuming that the number of data blocks is $L$ and the superblock size is $N$, the PGD size is equal to $c_1 \times L$ where $c_1$ is the size of each PGD entry. If the total number of PT cache entries is $n$ and the entry size is $c_2$, the cache size is given by $c_2 \times n$. For 32GB SLC NAND flash memory, $L$ is 262,144 (256K), $c_1$ is 3 bytes, and $c_2$ is 64 bytes. Assuming that Superblock FTL caches 16 entries, the total mapping information size becomes 769KB.

In case of MLC NAND flash memory with additional map pages, the PGD size is given by $(c_1 \times L)/(N \times k)$, where $k$ denotes the number of superblocks covered by a map page. The size of the map cache entry is equal to the page size. Thus, the size of mapping information is formulated by $(c_1 \times L)/(N \times k) + \mathit{PageSize} \times n$. For instance, consider 32GB MLC NAND flash whose page size is 4KB. When the superblock size is 4 ($N = 4$), $k$ is 6 and $c_1$ is 3 bytes. In this case, the PGD size becomes 8KB and the mapping information including the cache for 16 map pages requires 72KB. For SLC and MLC NAND flash whose capacity is 1TB, the memory consumption is about 24MB and 320KB, respectively.

## 6. CONCLUSIONS

In this article, we have proposed a novel FTL scheme called Superblock FTL for NAND flash memory. In Superblock FTL, the block-level mapping is still used at the superblock level, but logical pages within a superblock can be freely located in one of the physical blocks allocated to the superblock. This hybrid address translation scheme has the flexibility provided by fine-grain address translation, while reducing the memory overhead to the level of coarse-grain address translation. The notion of the superblock is effective in exploiting the block-level temporal and spatial locality, reducing not only the number of merge operations but also the average merge cost to make a free block. In addition, Superblock FTL makes use of spare areas in NAND flash memory to store page-mapping information so as not to incur any additional overhead in terms of space and flash memory operations.

From our results, the proposed FTL scheme significantly decreases the merge cost compared to previous block-mapped FTL schemes with roughly the same memory overhead. During the simulation study of representative storage access patterns, we also find out that it is very important to get storage access requests aligned on the page boundary of NAND flash memory.

## ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for valuable comments and Young-Sik Lee who helped the evaluation.

## REFERENCES

BAN, A. 1995. Flash file system. U.S. Patent, no. 5,404,485.

BAN, A. 1999. Flash file system optimized for page-mode flash technologies. U.S. Patent, no. 5,937,425.

BAPCO. 2007. SYSmark 2007 Preview. http://www.bapco.com/products/sysmark2007preview.

BITYUTSKIY, A. B. 2005. JFFS3 design issues. version 0.32 (draft). http://www.linux-mtd.infradead.org/tech/JFFS3design.pdf.

CHANG, L.-P. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the Symposium on Applied Computing (SAC)*. ACM, New York, 1126–1130.

CHANG, L.-P. AND KUO, T.-W. 2005. Efficient management for large-scale flash-memory storage systems with resource conservation. *ACM Trans. Storage 1,* 4, 381–418.

CHANG, L.-P., KUO, T.-W., AND LO, S.-W. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. Embedded Comput. Syst. 3,* 4, 837–863.

CHIANG, M.-L., LEE, P. C., AND CHANG, R.-C. 1998. Data management in a flash memory-based storage server. http://dspace.lib.fcu.edu.tw/bitstream/2377/2050/1/ce07ics001998000138.pdf.

CHIANG, M.-L., LEE, P. C. H., AND CHANG, R.-C. 1999. Using data clustering to improve cleaning performance for flash memory. *Software Pract. Exp. 29,* 3, 267–290.

CHOUDHURI, S. AND GIVARGIS, T. 2008a. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'08)*. ACM, New York, 19–24.

CHOUDHURI, S. AND GIVARGIS, T. 2008b. Real-time access guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'08)*. Springer-Verlag, Berlin, 138–149.

COOKE, J. 2007. Flash memory technology direction. In *Proceedings of the Windows Hardware Engineering Conference (WinHEC'07)*.

DAN, R. AND SINGLER, R. 2003. Implementing MLC NAND flash for cost-effectie, high-capacity memory. M-Systems Inc. http://www.data-io.com/pdf/NAND/MSystems/Implementing_MLC_NAND_Flash.pdf.

DOUGLIS, F., CACERES, R., KAASHOEK, F., LI, K., MARSH, B., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 25–37.

FUTUREMARK CORP. 2005. PCMark05. http://www.futuremark.com/products/pcmark05.

HARARI, E., NORMAN, R. D., AND MEHROTA, S. 1997. Flash EEPROM system. U.S. Patent, no. 5,602,987.

IDEMA. 2007. IDEMA Long Data Block White Paper. http://www.idema.org/_smartsite/ modules/local/data_file/show_file.php?cmd=standards&cat=103&h=1.

INOUE, A. AND WONG, D. 2003. NAND flash applications design guide. Tech. rep., Toshiba America Electronic Components, Inc.

INTEL CORP. 1998. Understanding the flash translation layer (FTL) specification. http://developer.intel.com/.

JUNG, D., CHAE, Y.-H., JO, H., KIM, J.-S., AND LEE, J. 2007. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. ACM, New York, 160–164.

KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th Annual ACM Conference on Embedded Systems Software (EMSOFT'06)*. ACM, New York.

KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash-memory based file system. In *Proceedings of the USENIX Winter Technical Conference*. USENIX, Berkeley, CA, 155–164.

KIM, J., KIM, J. M., NOH, S., MIN, S. L., AND CHO, Y. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. Consum. Electron. 48,* 2, 366–375.

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. Embedded Comput. Syst. 6,* 3.

LIM, S.-H. AND PARK, K.-H. 2006. An efficient NAND flash file system for flash memory storage. *IEEE Trans. Comput. 55,* 7, 906–912.

MICRON TECHNOLOGY INC. 2005. Small block vs. large block NAND flash devices. Tech. rep., Technical Note TN-29-07.

M. T. D. 2008. Ubi - unsorted block images. http://www.linux-mtd.infradead.org/doc/ubi.html.

PARK, C., CHEON, W., KANG, J., ROH, K., CHO, W., AND KIM, J.-S. 2008. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Trans. Embedded Comput. Syst. 7,* 4, 1–23.

PARK, C., CHEON, W., LEE, Y., JUNG, M.-S., CHO, W., AND YOON, H. 2007. A re-configurable FTL architecture for NAND flash-based applications. In *Proceedings of the 18th International Workshop on Rapid System Prototyping (RSP)*. IEEE, Los Alamitos, CA, 202–208.

PARK, C., SEO, J., SEO, D., KIM, S., AND KIM, B. 2003. Cost-efficient memory architecture design of NAND flash memory embedded systems. In *Proceedings of the International Conference on Computer Design (ICCD)*. IEEE, Los Alamitos, CA, 474–480.

RUEMMLER, C. AND WILKES, J. 1993. UNIX disk access patterns. In *Proceedings of the USENIX Winter Technical Conference*. USENIX, Berkeley, CA, 405–420.

RUSSINOVICH, M. 2006. DiskMon.
http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

SAMSUNG ELEC. 2003. 64Mx16 bit NAND flash memory (K9F1G16U0M).

SAMSUNG ELEC. 2006. 2Gx8 bit NAND flash memory (K9GAG08U0M).

SAMSUNG ELEC. 2007. 1Gx8 bit/2Gx16 bit NAND flash memory (K9WAG08U1A).
http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/16Gbit/K9WAG08U1A/ds_k9xxg08uxa_rev11.pdf.

WU, C.-H. AND KUO, T.-W. 2006. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design (ICCAD'06)*. ACM, New York, 601–606.

YIM, K. S., KIM, J., AND KOH, K. 2005. A fast start-up technique for flash memory based computing systems. In *Proceedings of the Symposium on Applied Computing (SAC'05)*. ACM, New York, 843–849.

Received December 2008; revised April 2009; accepted July 2009