A Superblock-based Flash Translation Layer for NAND Flash Memory *

Jeong-Uk Kang Heeseung Jo Jin-Soo Kim Joonwon Lee
Computer Science Division
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Korea
{ux,heesn}@calab.kaist.ac.kr {jinsoo,joon}@cs.kaist.ac.kr

ABSTRACT

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. This paper proposes a novel superblock-based FTL scheme, which combines a set of adjacent logical blocks into a superblock. In the proposed FTL scheme, 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 mapping technique 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 decreases the garbage collection overhead up to 40% compared to previous FTL schemes.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection*

General Terms

Management, Measurement, Performance, Design

Keywords

NAND flash memory, flash translation layer(FTL), address translation

1. INTRODUCTION

*This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

Many mobile devices, including MP3 players, PDAs (personal digital assistants), PMPs (portable media players), high-resolution digital cameras and camcorders, and mobile phones, demand a large-capacity and high-performance storage system in order to store, retrieve, and process large multimedia data quickly. In those devices, NAND flash memory is already becoming one of the most common storage medium because of its versatile features such as non-volatility, solid-state reliability, low power consumption, shock resistance, and high cell densities [4, 11].

NAND flash memory, however, has a restriction that a page, which is the basic unit of read and write operations, should be erased before being rewritten in the same location. This characteristic is sometimes called *erase-before-write*. Moreover, the erase operations can only be performed on a larger block than the page. Therefore, an intermediate software layer called a *flash translation layer* (FTL) is usually employed to hide the limitation of erase-before-write [8, 3]. FTL achieves this by redirecting each write request to an empty location in NAND flash memory that has been erased in advance, and by managing an internal mapping table to record the mapping information from the logical sector number to the physical location. Although FTL gives an ability to update the same logical sector transparently without intervention of erase operation, it needs extra flash memory operations to prepare empty locations and extra storage to maintain the internal mapping table. The amount of extra operations and storage required is drastically varied depending on the employed FTL schemes.

There is trade off between extra storage and extra operations. While coarse-grain address translation lowers the amount of extra storage, it may cause more extra flash memory operations to keep up the mapping state regularly. On the other hand, fine-grain address translation is flexible in handling of write requests smaller than a block, but demands more extra storage for managing mapping information. As the capacity of NAND flash-based storage increases, the extra storage required by fine-grain address translation actually imposes a serious cost problem in mass-market products [9].

In this paper, we propose a novel FTL scheme called *superblock FTL scheme* for NAND flash memory. In the proposed scheme, a superblock consists of a set of adjacent logical blocks. 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 extra flash memory operations, the fine-grain mapping information is stored in the spare area

of NAND flash memory. This hybrid mapping technique has the flexibility provided by fine-grain address translation, while reducing the memory overhead to the level of coarse-grain address translation. Performance evaluation based on a trace-driven simulation shows that our superblock scheme reduces the garbage collection overhead up to 40% compared to previous FTL schemes with roughly the same memory overhead.

The rest of the paper is organized as follows. The next section 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 scheme is presented. Section 5 compares the performance of our scheme with that of previous schemes based on a trace-driven simulation. Finally, our conclusions and future work are drawn in Section 6.

2. BACKGROUND AND RELATED WORK

In this section, we describe the characteristics of NAND flash memory and the differences between the small block and the large block NAND flash memory. Then, we present a short overview of FTL and 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 the main data area and 16 bytes of the spare area. The page is the basic unit of read and write operations in NAND flash memory. The spare area is often used to store management information and error correction code (ECC) to correct errors while reading and writing [6]. 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 management information and ECC to/from the spare area.

There are three basic operations in NAND flash memory: read, write (or program), and erase. The read operation fetches data from a target page, while the write operation writes data to a page. The erase operation resets all values of a target block to 1. 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. As the read and write operations are executed on a page basis, while the erase operation is performed on a much larger block basis, NAND flash memory is sometimes called a write-once and bulk-erase medium.

Unlike magnetic disks or other semiconductor devices such as SRAMs and DRAMs, the write operation requires a relatively long latency compared to the 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 – 1,000,000 times. Thus, the number of erase operations should be minimized to improve the overall performance and to lengthen 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 2 Kbytes of the main data area and 64 bytes of the 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 [5]. In this paper, we primarily focus on the large block NAND flash memory because most of the latest NAND flash devices whose capacity is more than 1 Gbits have the large block organization [7].

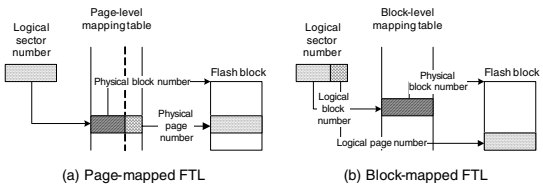

Figure 1: The address translation in FTL.

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 erase-before-write characteristics. Among others, two particularly important functions of FTL are address translation and garbage collection.

FTL hides the latency of erase operation by redirecting each write request from the host to an empty location in flash memory that has been erased in advance, and by managing the mapping information internally. The primary role of the address translation is to translate the logical sector number of a request into a physical address that represents a location of data in NAND flash memory. According to the granularity with which the mapping information is managed, FTLs are classified either as page-mapped [1] or as block-mapped [2]. Garbage collection is the process that reclaims free pages by erasing appropriate blocks.

A page-mapped FTL scheme is a fine-grained 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, a page-mapped FTL scheme allows for more flexible storage management. However, the size of the mapping table is 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.

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. A set of consecutive sectors in the logical block is usually 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 a block-mapped FTL scheme is significantly smaller compared to page-mapped FTL schemes.

As the capacity of NAND flash-based storage increases, the large amount of RAM required by a page-mapped FTL scheme actually imposes a serious cost problem in mass-market products. For example, a CompactFlash system with a 4 Gbytes large block NAND flash memory chip requires 8 Mbytes of RAM for maintaining the mapping ta-

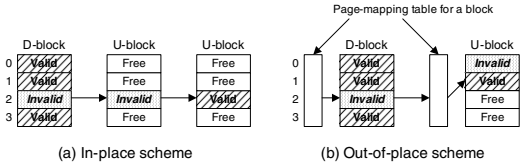

Figure 2: The page management schemes in a block.

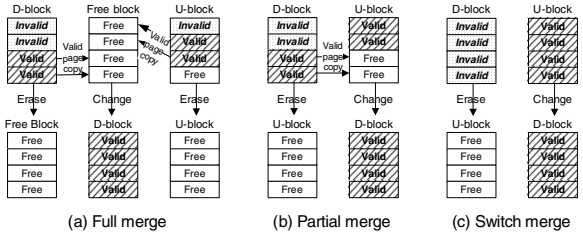

Figure 3: The types of merge operations during garbage collection.

ble with page-mapped FTL schemes, while 128 Kbytes for block-mapped FTL schemes. Thus, some variations of block-mapped FTL scheme are mostly 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, and 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 data 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 FTL schemes, the difference largely comes from the way those D-blocks and U-blocks are managed, i.e., when and how many U-blocks are allocated for each D-block, or how the merge operation is performed.

Logical pages in a D-block or a U-block are organized either by an *in-place* scheme or by an *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 in 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 (cf. 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 in large block NAND flash memory.

In the out-of-place scheme (cf. 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, garbage collection is invoked. During garbage collection, 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 (cf. 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 a block and two erase operations.

Partial and switch merges are special cases of the 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 invalidated, we can simply switch from the U-block to the new D-block and erase the old D-block. This case is called the switch merge (cf. Figure 3(c)). The switch merge requires only one erase operation without any valid page copy and hence is the optimal 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 files.

The performance of block-mapped FTL scheme significantly depends on how to organize D-blocks and U-blocks, and on how to select victim U-blocks during garbage collection. We note that the major causes of the performance degradation are due to valid page copy and erase operations to make free blocks during garbage collection.

2.4 Related Work

FTL schemes have been proposed to improve the performance in block-level mapping. Ban proposed the *replacement block scheme* based on the concept of a replacement block [2]. In this scheme, U-blocks are called replacement blocks, and both D-blocks and U-blocks are organized by an 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-blocks 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 subsection, the replacement block scheme exhibits poor storage utilization especially when only some of pages are frequently updated. Moreover, this scheme is not suitable for large block NAND flash memory, where pages in a block cannot be programmed in random order.

Kim et al. have suggested the *log block scheme* that uses U-blocks as logging blocks [9]. 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 an in-place scheme, while U-blocks by an 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 a 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 new free U-blocks. As D-blocks are managed by an in-place scheme, the full merge may happen in order to change from an out-of-place scheme to an in-place scheme. In addition, the utilization of the U-blocks can be still 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, the *fully associative sector translation (FAST) scheme* has been proposed [10]. In FAST, a U-block is shared by all the D-blocks, and every write request is logged into the current log block. This effectively improves the storage utilization of log 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 phenomenon, FAST uses the special U-block, called *sequential log block*, for handling sequential writes.

3. MOTIVATION

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

3.1 Rearranging pages in several blocks

The performance of FTL mainly depends on the efficiency of garbage collection. The overhead of garbage collection includes the time to erase blocks for making free blocks and the time to copy valid pages from to-be-erased blocks. To reduce the garbage collection overhead, many FTLs try to minimize the number of erase operations by maximizing the utilization of U-blocks. For example, unlike the replacement block scheme, the log block scheme uses an out-of-place scheme for U-blocks so that several updates to the same logical block can be absorbed in the 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 U-block.

We observe that, however, it is important to minimize not only the number of block erases, but also the number

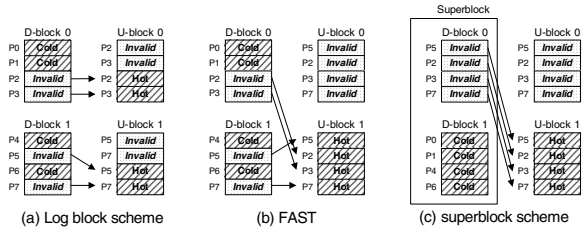

Figure 4: The situation where the full merge is occurred in the previous block-mapped FTL schemes.

of valid page copy. To reduce the time for valid page copy, we need to lower the number of full merge operations, while increasing chances of partial or switch merge operations.

In the previous block-mapped FTL schemes, full merges usually occur when pages within a block are randomly updated. To illustrate the problem, consider the situation shown in Figure 4. In this example, we assume that the number of physical pages per a block is four and the pages are updated in the following sequence: P5, P2, P3, P7, P5, P2, P3, and P7. Only two pages in each logical block are updated, namely P2, P3, P5, and P7.

In case of the log block scheme (cf. Figure 4(a)), to merge U-block 0 and U-block 1 with D-block 0 and D-block 1, respectively, two full merge operations are required. This is because there are not enough free pages in U-block 0 and U-block 1 to copy all valid pages in D-block 0 (P0 and P1) and in D-block 1 (P4 and P6). In FAST (cf. Figure 4(b)), while U-block 0 does not have any valid pages, two full merge operations for D-block 0 and D-block 1 are still required since U-block 1 has the pages that belong to both D-block 0 and D-block 1. FAST first merges D-block 0 with U-block 1 to generate the new D-block 0, and then merges D-block 1 with U-block 1 again for the new D-block 1.

If we can place all the updated pages to D-block 0 and other pages to D-block 1 as presented in Figure 4(c), we need only one switch merge operation between U-block 1 and D-block 0. The key observation is that if we can dynamically arrange the pages into the physical block, we can increase chances of partial or switch merge operations instead of the expensive full merge operation.

In our superblock FTL scheme, we define the superblock as a set of adjacent logical blocks that share D-blocks and U-blocks. We still use the block mapping at the superblock level, but we allow logical pages within a superblock to be freely located in one of the allocated D-blocks and U-blocks by maintaining the page-level mapping information within the superblock. During garbage collection, we try to separate hot pages from cold pages and put them into different D-blocks using the hybrid mapping technique.

3.2 Exploiting block-level spatial locality

observe that there are both block-level temporal locality and block-level spatial locality in typical storage access patterns. 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.

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

Another 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*. 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 chance of full merge operation. 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 occurrences of full merge operation can be kept low. In the proposed superblock scheme, we can easily control the degree of sharing by adjusting the superblock size.

4. SUPERBLOCK FTL SCHEME

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

4.1 Overall architecture

The basic idea of the superblock FTL scheme is to page-map N logical blocks into $N + M$ physical blocks. N is the number of logical blocks composing a single superblock, which is identical to the number of D-blocks allocated for the superblock. M is the number of U-blocks additionally allocated for the superblock. Therefore, N is determined by the superblock size, while M is dynamically changed according to the number of currently available U-blocks. If a new U-block is allocated to the superblock, M is increased by one. Meanwhile, M is decreased when garbage collection merges D-blocks with U-blocks.

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, belong to the superblock 0. When a write request arrives for a page in the superblock, the superblock scheme 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 for a superblock, the subsequent write requests to the superblock are logged in the U-block sequentially from the first page. This out-of-place scheme makes it suitable for use with large block 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.

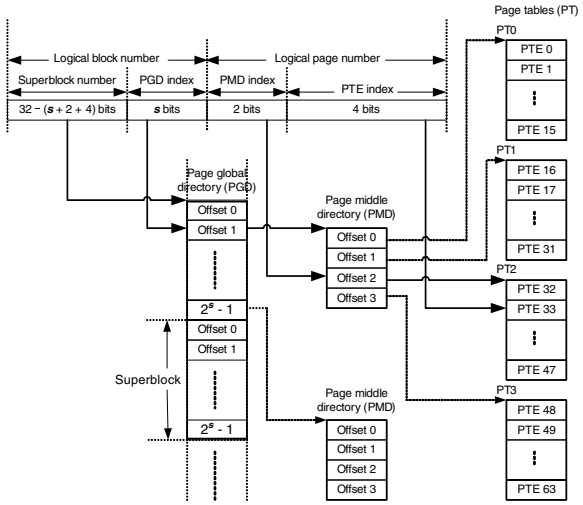

Figure 5: The address translation in the superblock scheme with three-level page-mapping table.

In order to make the superblock FTL scheme useful, we need to consider the following two questions: (1) how to maintain the mapping information compactly and efficiently, and (2) how to perform garbage collection intelligently to reduce the number of erase operations and valid page copy. In the following subsections, we attempt to answer these questions in detail.

4.2 Hybrid mapping with three-level mapping table

Since the superblock FTL scheme utilizes the page-level mapping inside the superblock, the pages belong to N logical blocks can be distributed anywhere in $N + M$ physical blocks. Therefore, maintaining the mapping information efficiently and compactly is a challenging issue.

We make use of spare areas to record the page-mapping information so as not to incur any additional overhead in terms of space and flash operations. 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.

We organize the page-mapping table in three levels as shown in Figure 5 so that it can be fit into the limited size of the spare area. This resembles the page table structure used in modern CPUs 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 four entries. Each PMD entry in turn points to the location of one of four page tables (PTs), whose entry (PTE: page table entry) contains the physical block number and the physical page number of the wanted data. Using the high 2 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 4 bits of the logical page number, PTE index.

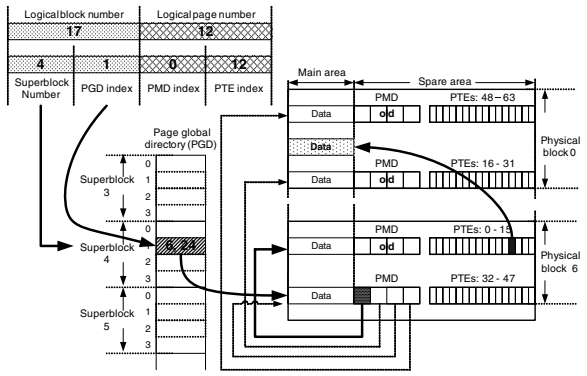

Figure 6: An example of the address translation in the superblock scheme.

Note that the whole page table is divided into four separate PTs due to the space limitation of the spare area within a single page in large block NAND flash memory. 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.

Figure 6 illustrates an example of address translation performed in the superblock scheme. 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, which holds PTEs from PTE0 to PTE15. Finally, 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 above 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 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 points to the valid page.

Since we should read PMD and the corresponding PT from flash memory every time when we read, write, or copy a page, we adopt a cache mechanism to reduce the number of flash read operations. A cache entry consists of PMD and the associated four PTs that are used to record the page-mapping information of a single logical block. The number of cache entries is fixed and we manage those entries with a least recently used (LRU) replacement scheme. This cache mechanism is similar to those used in the log block scheme

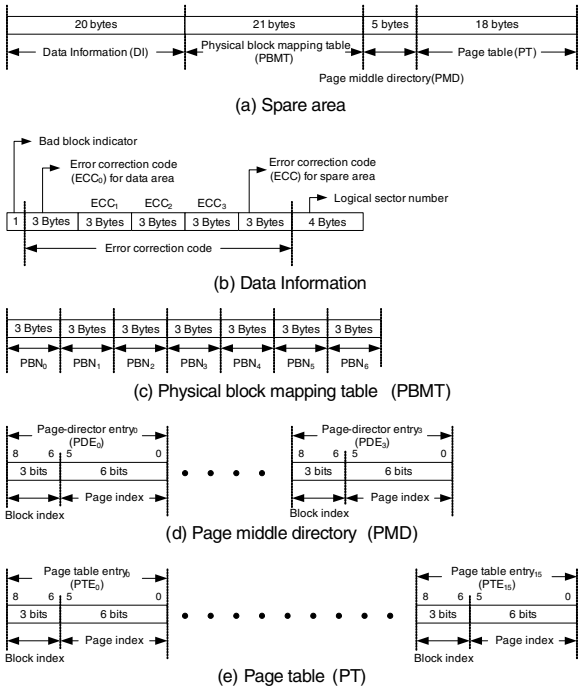

Figure 7: The format of the spare area in the superblock scheme in order to record the page-mapping information.

and FAST. Our experiment shows that the small number of cache entries works quite well (cf. Section 5.5).

Figure 7 depicts the overall layout of the spare area we use in the superblock scheme. 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 (cf. 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 (cf. Figure 7(d)), and each PT consists of 16 PTEs (cf. 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 can be fit into the spare area. In our superblock scheme, PBMT has an array of actual physical block numbers allocated for the 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 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

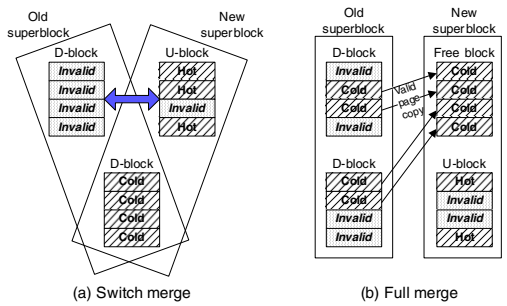

Figure 8: Two examples of merge operations in the superblock scheme.

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 number of D-blocks and U-blocks that can be allocated to a superblock is limited to eight in our current implementation.

4.3 Garbage collection

We need an intelligent garbage collection mechanism in order to reduce the number of erase operations and valid page copy, which are the major causes of performance degradation. Garbage collection is invoked if there is no free U-block to allocate. During garbage collection, a physical block is selected as a victim and then it is merged with other block to make a free block. The detailed process of garbage collection is as follows.

The first step is to find a physical block that has no valid pages. If there is such a block, it is erased and then allocated to another superblock. In case the chosen block is a D-block, one of U-blocks is promoted to D-block (switch merge), as shown in Figure 8(a). In this step, we only examine those superblocks that have at least one U-block, since investigating all the physical blocks is time-consuming, hence impractical.

If the first step fails, we find the superblock that has the least recently written U-block. In case there is the D-block that has sufficient free pages, we perform the partial merge. In the other case, we do not merge the U-block with one of D-blocks in order to separate hot pages from cold pages. Instead, we select two D-blocks from the superblock, which has the smallest number of valid pages, and perform the full merge, as illustrated in Figure 8(b). The rationale behind this decision is that U-blocks tend to have hot pages, while D-blocks cold pages. Therefore, putting cold pages stored in D-blocks together into a free block is desirable for future garbage collection. After the full merge, the superblock is reorganized in such a way that the new block and the U-block become D-blocks, and the original D-blocks are erased and reused as free U-blocks. In this way, we can achieve the effect that the pages of a superblock with similar update frequency flock together into the same D-block. Since the number of valid pages contained in two D-blocks may exceed the number of pages that can be stored in a free block, the full merge operation is continued until all the valid pages are moved and we can ultimately obtain a free block.

Table 1: Traces used for simulation

Trace	Description	Total storage size (MB)	The number of pages written
PIC	This trace models the workload of digital cameras. Picture files whose average size is 1 Mbytes are created and deleted.	2048	1,121,352
MP3	This trace models the workload of MP3 players. MP3 files whose average size is 5 Mbytes are created and deleted.	2048	1,437,522
MOV	This trace models the workload of movie players. Movie files whose sizes vary from 15 to 30 Mbytes are created and deleted.	2048	1,832,613
PMP_2G	This trace models the workload of portable media players (PMPs). A number of picture files, MP3 files, and movie files are created and deleted.	2048	1,222,218
PMP_4G		4096	2,403,474
PC	This trace is extracted from a real user activity on the notebook of personal usage during one week.	4096	2,724,393

As a special case, the least recently written U-block selected in the previous step may have some free page slots. This happens when the superblock includes only one U-block. In this case, most pages are cold and stored in D-blocks, while only a small number of hot pages stay in the U-block. Since all the D-blocks are almost full, performing the full merge operation between two D-blocks incurs high overhead, and we have no choice but to partially merge the U-block with one of D-blocks.

As described in the previous subsection, the number of U-blocks and D-blocks allocated to a superblock is restricted to eight due to the restriction in maintaining the mapping information. Thus, when we are to allocate the ninth physical block to a superblock, the same garbage collection algorithm is performed for the superblock to reclaim one of the physical blocks.

5. PERFORMANCE EVALUATION

This section evaluates the performance of the proposed superblock FTL scheme. For comparison, we have also evaluated two previous block-mapped FTL schemes, the log block scheme and FAST.

5.1 Evaluation methodology

We have implemented trace-driven simulators for the superblock scheme, the log block scheme, and FAST. The

Table 2: Access times large block NAND flash memory

Operation	Access time
NAND Flash read time (μs/page)	129.72
NAND Flash write time (μs/page)	298.88
NAND Flash erase time (μs/block)	1,998.70

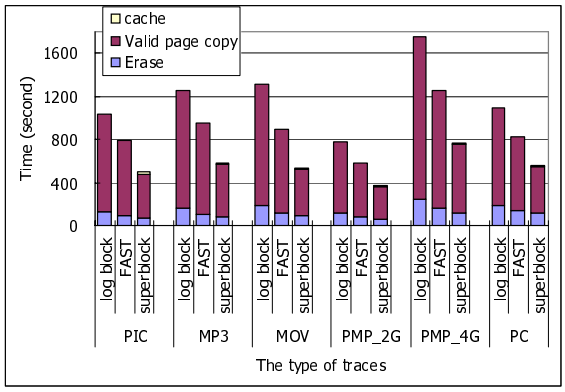

Figure 9: The garbage collection overhead of the log block scheme, FAST, and the superblock scheme.

workload is chosen to reflect the representative storage access patterns for notebook computers and multimedia mobile devices. Table 1 summarizes the characteristics of traces used in this paper. These traces are extracted from disk access logs of real user activities on FAT32 file system. Three traces, PIC, MP3, and MOV, model the workload of digital cameras, MP3 players, and movie players, respectively. We also model the workload of portable media players (PMPs) by creating and deleting various picture files, MP3 files, and movie files. The PC trace is the storage access trace of a real user during one week, which includes web surfing, word processing, presentation, and playing games, MP3 songs, and movies.

The sizes of NAND flash memory experimented are 2 Gbytes and 4 Gbytes. The simulators model the parameters related to current technologies as exactly as possible. The access times of large block NAND flash memory are summarized in Table 2. These parameters are actually measured on Samsung NAND flash chip model K9F1G16U0M.

The main performance metrics we use are the number of erase operations and the number of valid pages copied during garbage collection since they are the major factors limiting the performance of FTL. The garbage collection overhead is calculated based on the parameters shown in Table 2.

5.2 Overall performance

Unless otherwise stated, we performed our experiments in the following conditions. The superblock size is four (N=4), and the number of cache entries is 16. The number of available U-blocks is 3.1% of the number of D-blocks, hence the number of D-blocks is 16,384 and that of U-blocks is 512 in 2 Gbytes large block NAND flash memory.

Figure 9 depicts the garbage collection overhead for each block-mapped FTL scheme. There are three bars for each trace, which corresponds to the log block scheme, FAST, and the superblock scheme, respectively. We breakdown the garbage collection overhead according to the time spent on cache manipulation, valid page copy, and erase operations.

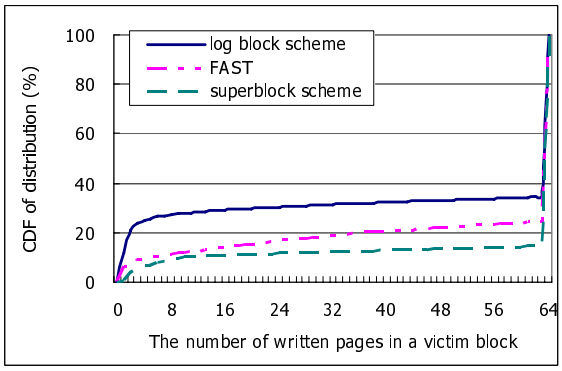

Figure 10: The cumulative distribution function (CDF) of the number of written pages in each victim block (PC trace).

Overall, the superblock scheme exhibits noticeably shorter garbage collection overhead than other schemes. The superblock scheme outperforms FAST by reducing the garbage collection overhead by 32% – 40% over the whole trace. In particular, we can observe that most of the benefits come from the decrease in the number valid pages copied during garbage collection; the superblock scheme reduces the time spent on valid page copy by 37% – 44% compared to FAST.

Notice that the cache manipulation time in the superblock scheme is negligible. This also includes the time to manage the page-mapping information stored in three-level mapping tables, but the overhead is only 2.3% – 2.9% of the total garbage collection overhead. From these results, we can see that the superblock scheme is indeed quite effective in reducing the garbage collection overhead, while imposing very little management overhead. As varying the trace does not change the trend of overall performance, we only show the results for PC trace in the following discussions.

The storage utilization is an important factor that affects the performance of FTL, since the lower storage utilization leads to more frequent garbage collection. To investigate the impact of each FTL scheme on the storage utilization, we have measured the number of written pages in each U-block, when the block is selected as a victim. Figure 10 illustrates the cumulative distribution function (CDF) of the measured values for PC trace.

Figure 10 shows that most of victim blocks are either full or occupied by less than 4 pages. Because a physical block has 64 pages in large block NAND flash memory, the value for 64 pages in Figure 10 indicates the percentage of the blocks that was full of written pages. For the log block scheme, about 65% of the blocks were fully used when they were selected as victims during garbage collection. As expected, FAST shows the better storage utilization compared to the log block scheme due to the increased degree of sharing; the percentage of the fully used blocks is increased to 75% in FAST.

Note that the percentage of the fully used blocks for the superblock scheme is about 85%, which is significantly better than that of FAST. This is very interesting because U-blocks

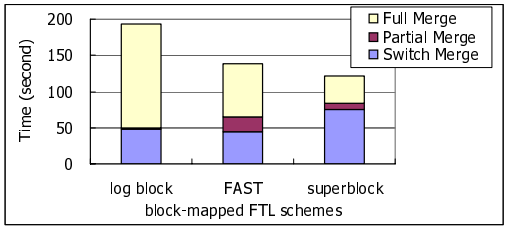

Figure 11: The ratio of erase operations by each merge operations (PC trace).

are shared by all the logical blocks in FAST, while they are shared only by the logical blocks within the same superblock in the superblock scheme. Our measurement shows that this is caused by a shortcoming inherent in FAST. FAST has sequential log blocks to optimize for a sequential write pattern. Unlike the other (random) log blocks, these sequential log blocks are assigned to a dedicated logical block as same as the log block scheme. In case the prediction of the sequential access pattern is wrong, the sequential log block can be merged even though the block is not full. The superblock scheme does not have such a problem, since several adjacent logical blocks in a superblock always share a U-block. Therefore, we can see that the proposed superblock scheme is a more effective and more robust way of exploiting the block-level spatial locality.

Figure 11 compares the execution time spent on erase operations in detail according to the type of the merge operation: full merge, partial merge, and switch merge.

We can find that, in the superblock scheme, 72% more erase operations are caused by the switch merge compared to FAST. This is because the superblock scheme shares D-blocks and U-blocks among several logical blocks and organizes all physical blocks with an out-of-place scheme, which increases the chance of the switch merge. On the other hand, erase operations caused by the full merge are significantly reduced in the superblock scheme. When the pages are not sequentially aligned in a U-block, the log block scheme and FAST need to perform the full merge operation to maintain D-blocks with an in-place scheme. However, the superblock scheme can switch the U-block to a new D-block, simply converting the full merge into the switch merge.

5.3 The effect of the number of U-blocks

Figure 12 shows the garbage collection overhead for each block-mapped scheme when the amount of U-blocks is varied from 16 (0.05% of the number of D-blocks) to 2048 (6.25% of the number of D-blocks). Again, three bars correspond to the results of the log block scheme, FAST, and the superblock scheme, respectively. As the amount of U-blocks is raised, the garbage collection overhead of all schemes is lowered. This is an expected result since the chance of garbage collection decreases as there are more free blocks.

When the amount of U-blocks is 16, FAST indicates the better performance than the superblock scheme. The superblock scheme cannot exploit the advantage of block-level temporal and spatial locality with a small number of U-blocks. This leads to more frequent garbage collection due to the lower storage utilization. However, as the amount of U-blocks grows, the superblock scheme shows the much better performance than FAST.

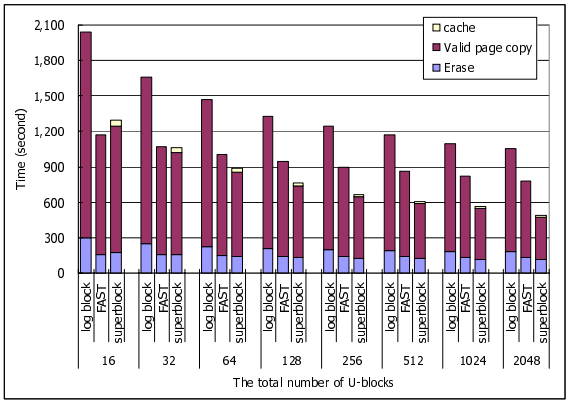

Figure 12: The impact of the number of U-blocks on the garbage collection overhead (PC trace).

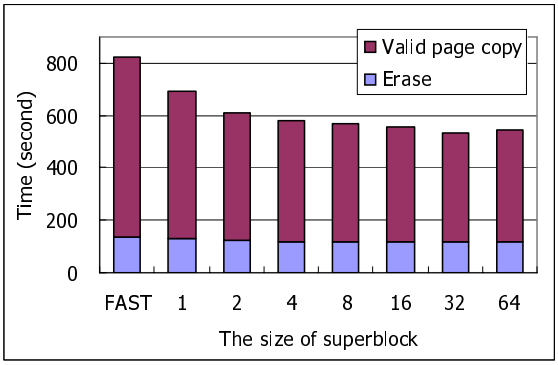

Figure 13: The impact of the superblock size on the garbage collection overhead (PC trace).

5.4 The effect of the superblock size

Figure 13 investigates the impact of the superblock size on the garbage collection overhead. For this experiment, we held all the page-mapping information in RAM without storing them in the spare area, since the current implementation cannot support the superblock size greater than eight.

Note that the garbage collection overhead is reduced by 16% compared to FAST even when the superblock size is one. The performance gain is largely resulted from organizing D-blocks with an out-of-place scheme.

As superblock size grows from 1 to 32, the garbage collection overhead is decreased because the storage utilization of U-blocks increases due to the block-level spatial locality. This reduces the number of garbage collection invoked. When the superblock size is 32, the garbage collection overhead is decreased by 23% compared to the result with the superblock size 1.

However, from the case that the superblock size is 64, the garbage collection overhead increases slowly. This is because the larger degree of sharing tends to increase the chance of full merge operation and to cause more valid page copies and erase operations to make a free block. In the worst case, we must relocate all valid pages in a superblock. Figure 13 shows that increasing the superblock size above 4 has only marginal benefits.

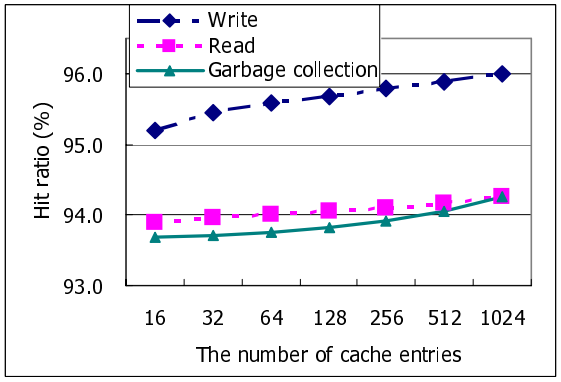

Figure 14: Changes in cache hit ratios according to the number of cache entries in the superblock scheme (PC trace).

5.5 The effect of the cache size

Figure 14 presents the change of the cache hit ratios when the number of cache entries increases from 16 to 1024. Note that the hit ratio of the smallest cache size is more than 93% for all types of requests. This shows that the block-level temporal locality and page-level spatial locality are very high in the tested workload.

As the number of cache entries increases from 16 to 1024, the improvement of hit ratio is 1.2% at most (note the scale of the y-axis). Therefore, 16 cache entries are sufficient in most cases.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel FTL scheme for NAND flash memory, which we call a superblock FTL scheme. In the superblock scheme, we still use the block mapping at the superblock level, but we allow logical pages within a superblock to be freely located in one of the allocated physical blocks. This hybrid mapping techniques has the flexibility provided by fine-grain address translation, while reducing the memory overhead to the level of coarse-grain address translation. The superblock scheme reduces the number of garbage collections invoked, as well as increases chances of switch merge operations instead of expensive full merge operations. The hybrid mapping technique makes use of spare areas in NAND flash memory to store page-mapping information so as not to incur any additional overhead in term of space and flash memory operations.

In addition, using the notion of the superblock is quite effective in exploiting both of the block-level temporal locality and the block-level spatial locality. Especially, the superblock FTL scheme can easily control the degree of sharing by adjusting the superblock size according to the block-level spatial locality. As the degree of sharing increases, the chance of garbage collection falls due to the increased storage utilization, but the chance of the full merge operation rises. In our experiment, the superblock size of 2 or 4 worked very well in most cases.

From our results, the proposed FTL scheme decreases the garbage collection overhead up to 40% in compared to previous FTL schemes with roughly the same memory overhead.

For future work, we are going to design a mechanism, which constructs the first-level mapping table (PGD) in RAM as quickly as possible for fast startup. We also plan to investigate how to achieve the power-off recovery effectively under our FTL scheme.

7. REFERENCES

[1] C. Association. http://www.compactflash.org/.

[2] A. Ban. Flash file system. United States Patent, no. 5,404,485, April 1995.

[3] I. Corporation. Understanding the flash translation layer (ftl) specification. http://developer.intel.com/.

[4] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI-1)*, pages 25–37, November 1994.

[5] S. Electronics. 1g x 8 bit / 2g x 16 bit nand flash memory. Available from: <http://www.samsung.com/Products/Semiconductor/NAND-Flash/SLC_LargeBlock/16Gbit/K9WAG08U1M/K9WAG08U1M.htm>, 2005.

[6] E. Harari, R. D. Norman, and S. Mehrota. Flash eeprom system. United States Patent, no. 5,602,987, February 1997.

[7] M. T. Inc. Small block vs. large block nand flash devices. Technical report, Technical Note TN-29-07, 2005.

[8] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 155–164, 1995.

[9] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[10] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S.-W. Park, and H.-J. Songe. FAST: A log-buffer based ftl scheme with fully associative sector translation. In *The 2005 US-Korea Conference on Science, Technology, & Entrepreneurship*, August 2005.

[11] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. In *Proceedings of the 21st International Conference on Computer Design (ICCD '03)*, pages 474–480, October 2003.