A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications

CHANIK PARK, WONMOON CHEON, JEONGUK KANG,
KANGHO ROH, and WONHEE CHO
Samsung Electronics
and
JIN-SOO KIM
Korea Advanced Institute of Science and Technology

In this article, a novel FTL (flash translation layer) architecture is proposed for NAND flash-based applications such as MP3 players, DSCs (digital still cameras) and SSDs (solid-state drives). Although the basic function of an FTL is to translate a logical sector address to a physical sector address in flash memory, efficient algorithms of an FTL have a significant impact on performance as well as the lifetime. After the dominant parameters that affect the performance and endurance are categorized, the design space of the FTL architecture is explored based on a diverse workload analysis. With the proposed FTL architectural framework, it is possible to decide which configuration of FTL mapping parameters yields the best performance, depending on the differing characteristics of various NAND flash-based applications.

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

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, FTL, reconfigurable architecture, performance analysis

ACM Reference Format:

Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., and Kim, J. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. ACM Trans. Embedd. Comput. Syst. 7, 4, Article 38 (July 2008), 23 pages. DOI = 10.1145/1376804.1376806 http://doi.acm.org/10.1145/1376804.1376806

Authors' addresses: C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, Samsung Electronics, Hwasung-City, Korea; email: {ci.park, wm.cheon, ju.kang, kangho.roh, whpp.cho}@samsung.com; J.-S. Kim, Division of Computer Science, Korean Advanced Institute of Science and Technology, Daejeon, Korea; email: jinsoo@cs.kaist.ac.kr.

1. INTRODUCTION

NAND flash memory has become more common in many mobile devices, such as MP3 players, MMC cards, cellular phones, and PDAs, as it is nonvolatile, reliable, uses relatively less power, and is more resistant to physical shocks. As the cost per bit has continuously decreased, NAND flash-based solid-state drives are penetrating the laptop PC market as a complementary medium or a competitive replacement to the magnetic disks that have been used since their introduction [Min 2004].

However, unlike magnetic disks, NAND flash memory is characterized by its *erase-before-write* operation; it must be erased before new data is written to a given physical location. This inherently necessitates NAND flash management software known as an FTL (flash translation layer), which handles the algorithmic sequences of read, write, and erase operations of NAND flash. The FTL receives read and write requests and maps a logical address to a physical address in NAND flash.

Although a key role of the FTL is to hide the technological details of NAND flash and to maximize the performance and lifetime of the underlying storage device, it is designed and implemented with different constraints, depending on each target application. In the case of embedded storage architecture for mobile devices (Figure 1(a)), the FTL is implemented as a block device driver below file systems, such as FAT16/32. As a result, its interoperability with file systems and the operating system is an important development issue aside from performance optimization. Despite the fact that the memory and computation power restrictions may be relatively less tight because of the abundant DRAM and high-performance CPU of the host system, emerging multimedia applications require even higher storage performance. On the other hand, in removable storage architecture, such as MMC (multimedia card) and UFD (USB flash drive) (Figure 1(b)), the FTL is implemented into the firmware for an onboard low-cost embedded controller. The read/write performance can be maximized with the assistance of dedicated hardware, while limited computation power and memory resources should be considered when implementing the FTL.

Another important factor that affects FTL design is the access patterns of different applications. In particular, the sequential or random-access behavior and the length of the requested data dominate the performance of FTL. For example, in the case of a recording application implemented by a digital camera or camcorder, logical-address accesses are characterized by short random and long sequential patterns, as shown in Figure 2. The analysis shows that the short random-access pattern results from access to the file system's metadata (FAT, directory entry) and the long sequential pattern is caused by user data (recorded image).

NAND flash storage embedded in cellular phones and PCs tends to show comparatively more random-access patterns than sequential types as multiple access requests from concurrently executed applications typically occur. Consequently, the access patterns of applications are another consideration for an efficient FTL design.

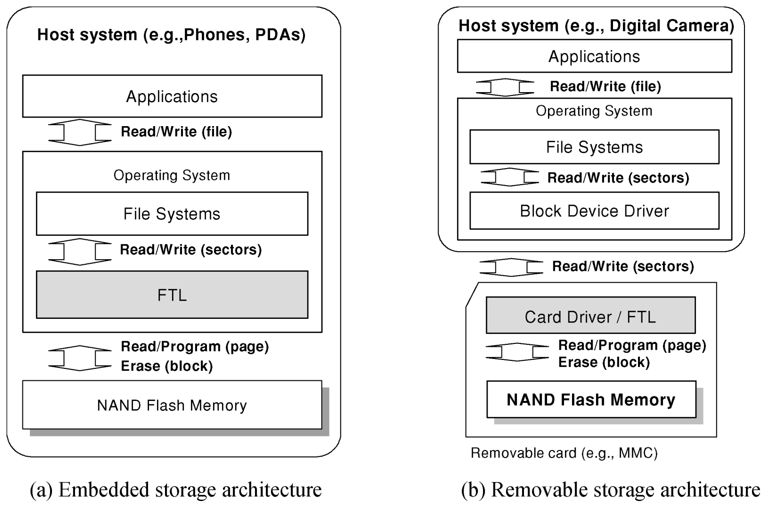


Fig. 1. Software architectures for NAND flash-based applications.

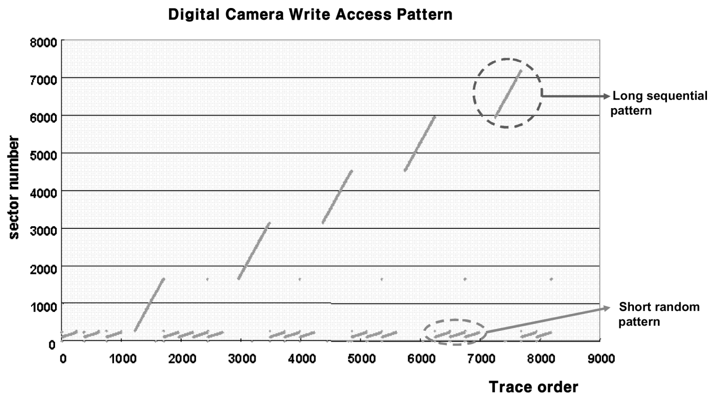


Fig. 2. An example of a workload trace from a digital camera.

As the application area of NAND flash has been widened, a flexible FTL architecture to cope with various requirements from NAND flash applications has become a main concern for system designers who need to satisfy time-to-market delivery requirements coupled with performance and memory constraints.

In this article, a reconfigurable FTL architecture is proposed that aims at building an FTL that is optimized for each target NAND application in terms of its performance, endurance, and memory requirements. The proposed FTL architecture is based on a flexible mapping structure configured using the two design parameters of the spatial and temporal locality of target applications. In order to find the optimal parameter values, intuitive, but efficient, workload analysis is initially performed so that the design space can be narrowed down without exhaustive exploration of the parameters. In addition, a formal model of the performance and memory requirements will provide FTL designers with an analysis tool for correlating the pattern and FTL design parameters of target

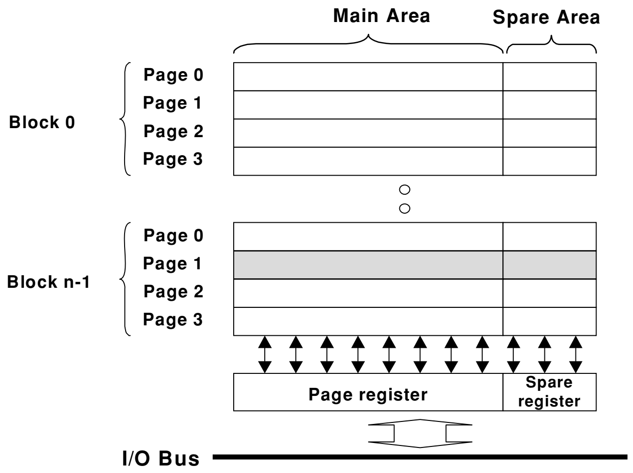


Fig. 3. NAND flash structure.

applications. The usefulness of the proposed FTL architecture is verified in tests involving actual MP3 and PC applications.

The remainder of this article is organized as follows. Section 2 gives a brief overview of NAND flash memory and typical FTL concepts. Section 3 discusses related work and the motivation for the proposed FTL architecture. In Section 4, a detailed description of the proposed reconfigurable FTL architecture is presented. Section 5 introduces the performance model of the proposed FTL architecture, in addition to an analysis of it. Finally, the experimental results are given and the conclusions are presented in Sections 6 and 7, respectively.

2. BACKGROUND

2.1 NAND Flash Structure

A NAND flash memory component consists of a fixed number of blocks with each block consisting of 64 pages and each page consisting of 2 KB of main data and 64 bytes of spare data. This is shown in Figure 3. Read and write operations are performed on a page basis, while an erase operation is executed on a block basis. In order to read a page, the command code and page address are inputted to the NAND flash memory through I/O pins. After the "Page Read" latency (refer to Table I), the selected page is loaded into the page and spare registers. Finally, the loaded data is transferred to the system memory through the I/O bus. Spare data can be used to store auxiliary information, including that termed bad-block identification and error-correction code (ECC) for the associated main data. For a write operation, a command code and a page address are issued and data is loaded from the system memory to the page register and the spare register. After the "Page Program" latency, the data is programmed into the designated page. For an erase operation, the command code and block address are inputted. After the "Block Erase" duration, the corresponding block is erased.

Table I. Operation Latency of NAND Flash[a]

Operation	Latency
Page read	20 us
Page program	200 us
Block erase	1.5 ms

[a]Samsung Electronics [2005].

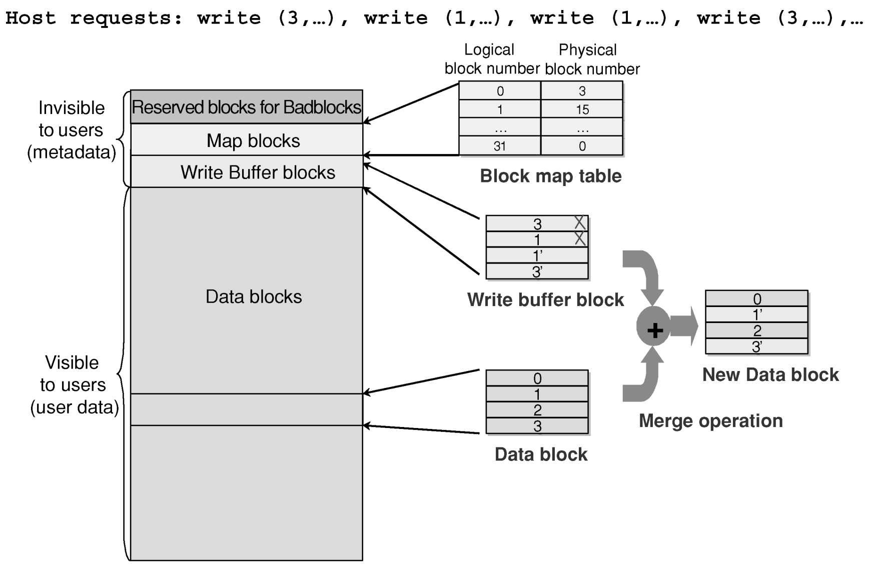


Fig. 4. Logical view of the FTL of NAND flash memory.

Unlike magnetic disks or other semiconductor devices, such as SRAMs and DRAMs, a write operation requires a relatively long latency compared to a read operation. In addition, as a write operation may accompany an erase operation, the write 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 approximately 100,000. Thus, the numbers of write and erase operations should be minimized not only to improve the overall performance but also to maximize the lifetime of NAND flash memory.

2.2 FTL Concepts

A typical FTL will logically divide the NAND flash into a meta data area and a user data area. The metadata area includes Reserved blocks for replacing initial or run-time bad blocks, Map blocks for translating logical to physical addresses, and Write buffer blocks for temporarily storing the incoming write data (Figure 4). A fixed number of data blocks constitute the user data area in which the user data resides.

In a situation in which a block consists of four pages and that a sequence of page-write requests to the logical addresses (3, 1, 1, 3) occur (Figure 4), the logical address is initially translated into a physical address based on the block map table. In this example, all logical addresses happen to belong to the same data block. The data block is occupied by previous data and an overwrite operation to NAND flash is not allowed; hence, a temporary block known as a "write buffer block" is allocated to store the incoming data pages. After the first two write requests to the logical addresses (3, 1) are performed on the first and second pages of the write buffer block, the following overwrite requests to logical addresses (1, 3) are stored in the third and fourth pages, as an in-place update is prohibited in flash memory. As a result, the first and second pages are marked as invalid data and the third and fourth pages are kept as valid data. If there are additional write requests while the write buffer block has no free pages to write (for simplicity, here the number of write buffer blocks is considered to be one), new free pages should be produced. To make space for a new write request, the fully written write buffer block is reclaimed by merging it with the corresponding data block. During the merge operation, a new data block that is sourced from free blocks is allocated and valid pages are copied from the write buffer block and the data block into the new data block. In this case, the merge operation requires four page-read operations, four page-program operations, and two block-erase operations (one for the write buffer block and the other for the data block).

The merge operation consumes a considerable amount of time; therefore, reducing the numbers of merge operations and the required read/program operations at each merge operation is a main concern with FTL performance. To address this problem, several mapping schemes have been suggested [Chang and Kuo 2002, 2004, Kim et al. 2002, Gal and Toledo 2005, Kang et al. 2006].

3. MAPPING SCHEMES AND RELATED WORK

The logical to physical address translation is based on a mapping scheme. There are two types of mapping schemes depending on the granularity with which the mapping information is managed: block and page mapping.

In page mapping [Chiang et al. 1999], NAND flash memory is managed on a page basis. Therefore, a page map table is constructed and maintained in both NAND flash memory and RAM. A map table entry consists of an LPN (logical page number) and a PPN (physical page number). When a write request is sent to some logical page address, the corresponding physical page number is located using the page map table. If the page found is at that juncture written with some value, the page is invalidated and the requested data is written to an available free page.

As an example (see Figure 5), when a write request to logical page address 5 is inputted to the FTL, the FTL first searches for the corresponding physical page number using the logical page number, both of which have the same index in the page map table. As a result of the matching of the address, the corresponding physical page number was found to be 2 in the NAND flash memory. However, the physical page number 2 is, in this case, occupied with valid data. Hence, the requested data should be written to a free page in flash memory. As the second page of physical block number 1 (physical page number 5) is free, the data

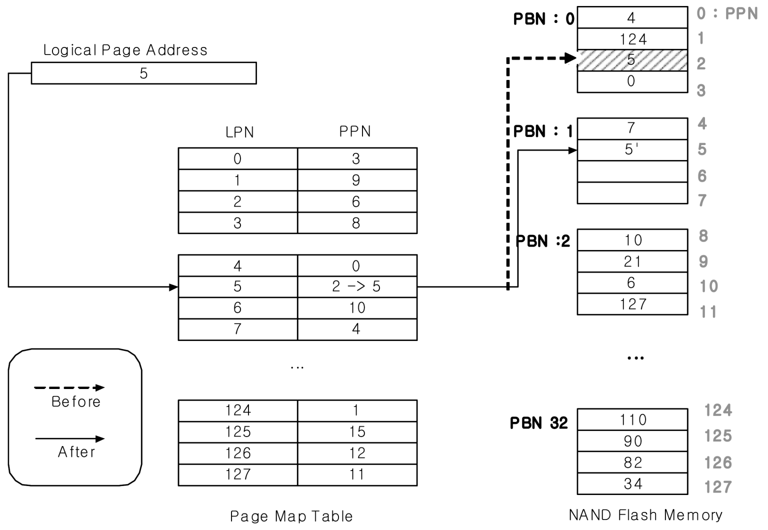


Fig. 5. Page-mapping scheme.

is written to that location. At the same time, the corresponding map entry is updated to point to the new page with valid data. The page-mapping scheme has an advantage in that it writes data to any free page in flash memory, which adds flexibility to storage management. Therefore, random write-access patterns can be accommodated without frequent block reclamation processes that involve a number of page-copy operations and block-erase operations. Although the page-mapping scheme shows better performance when enough free pages are available, the invalid pages should be reclaimed as their numbers increase in order to make free space available for new data. In this case, performance can degrade drastically. Therefore, an efficient "garbage collection" technique should be devised. Another problem with page mapping is that it requires a very large amount of memory space (in both RAM and flash memory) for the map table. For instance, assuming that the flash memory has a density of 512 MB, a map table size of 1 MB is required. As 1 MB of RAM is not viable in cost-competitive embedded systems, such as flash memory card storage applications, a map table caching scheme can be adopted at some performance cost.

In order to reduce the map table size, block mapping can be utilized. In block mapping [Ban 1995], the logical page address is divided into a logical block number and a page offset. The logical block number is used to find a free block that includes free pages, and the page offset is used as an offset to locate the free page in the corresponding block. As the map table consists of block number entries, its size can be reduced from 1 MB to 16 KB as a block consists of 64 pages [Samsung Electronics 2005]. Thus, this scheme can be accommodated in a diverse range of embedded systems resulting from its efficient use of memory.

However, given that the page offset is extracted from the logical page address of the host, the page offsets of the logical and physical blocks should be identical. As a result, every overwrite operation to the same logical page may incur a frequent block-level copy operation.

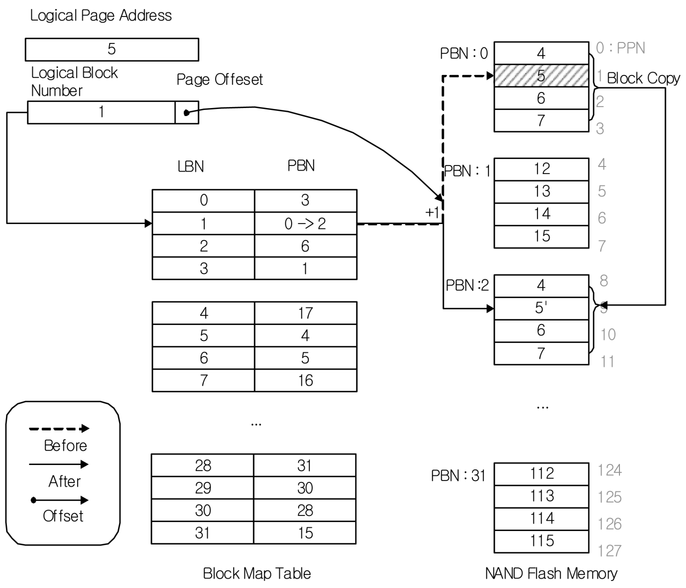


Fig. 6. Block-mapping scheme.

For example (see Figure 6), when a write request to logical page address 5 is inputted to the FTL, the logical page address is divided into logical block number 1 and page offset 1. The physical block number for the corresponding logical block number 1 is determined first. After the corresponding physical block number 0 is matched, the logical page offset is added to the determined physical block number, and the incoming data is then written to physical page number 1. However, in this case, physical page number 1 has already had data written to it. Therefore, the data should be written to a free block (physical block number 2). At the same time, the other pages in the same block where physical page number 1 is located are copied to the same free block as one logical block is associated with only one physical block in this scheme. Block mapping yields better performance over sequential write-access patterns, though it may show considerable performance degradation over random-access patterns.

As an approach that compromises between page mapping and block mapping, many hybrid mapping schemes were proposed to reduce not only the mapping table size, but also the block copy overhead. A hybrid mapping scheme known as the log block scheme was first presented by Kim et al. [2002]. The key idea of the log scheme is to maintain a small number of log blocks in flash memory to serve as write buffer blocks for overwrite operations (hereafter, a write buffer block is referred to as a log block as their purposes are identical). The log block scheme allows the incoming data to be appended continuously as long as free pages are available in the log blocks. When an overwrite operation occurs with the same logical page data, the incoming data is written to a free page and the previous data becomes invalid.

For example, in Figure 7, which assumes that the physical block number 0 contains the data of logical page numbers (4, 5, 6, 7), when upcoming write

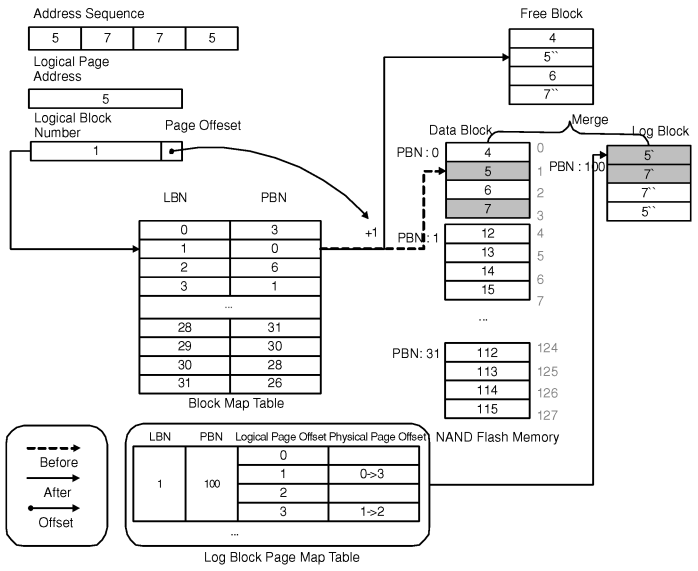


Fig. 7. Hybrid mapping scheme.

requests are to logical page numbers (5, 7, 7, 5), they are written to the allocated log block. The final two writes are overwrites for the first two writes. As a result, only the last requests to logical page numbers (7, 5) are valid for logical block number 1. These requests are represented as (5″, 7″) in the figure. When a log block has no additional free pages or when the logical block that contains the requested page is changed from the previous logical block, the log block and the corresponding data block are merged into a free block, as shown in Figure 7. Finally, the merged free block becomes the new data block, and the original data block and the log block become two free blocks. The free block map table is omitted in the figure for simplicity.

The log block scheme efficiently deals with both sequential and random writes. If there is a write request, it writes the data into a log block sequentially and maintains the separate page mapping information only for the log blocks. As only the small number of log blocks is used by the FTL, the amount of mapping overhead that is added is low. When all of the log blocks are used and a new write request comes to a data block, which is not associated with any log block, one of the log blocks in use should be merged with the corresponding data block to create writable free space. Thus, the log block scheme may experience a low utilization of the log blocks as a single log block is associated with only a single data block. As a result, the number of costly merge operations will increase with the quantity of unused free pages.

To solve this problem of the log block scheme, the fully associative sector translation (FAST) scheme has been proposed [Lee et al. 2006]. In FAST, a log block is shared by all of the data blocks, and every write request can be written into the current log block. This effectively improves the storage utilization of log blocks and greatly delays the merge operation. However, merging may be performed more frequently than in previous schemes as a single log block contains pages that are associated with several data blocks. To offset this phenomenon, FAST dedicates a special log block known as a sequential log block to handle sequential writes. In particular, FAST may suffer from a longer merge operation time which should be avoided in real-time constrained applications, such as voice recording.

Chang and Kuo [2004] proposed a flexible management scheme for large-scale flash-memory storage systems. It manages a high-capacity flash memory with different granularity sizes as differently sized leaves of a buddy tree. Their main goal is to obtain the flexibility of page mapping while requiring less memory. In contrast to the page- and block-mapping scheme, which both involve a fixed-size mapping unit, the scheme of Chang and Kuo utilizes mapping units of variable sizes. Several experiments were performed to demonstrate the reduction of the RAM requirements, the performance improvement, and the lengthening of the flash-memory lifetime in comparison with the simple page- or block-mapping schemes. The effectiveness of their scheme, however, is dependent on the pattern of the workloads. Hence, in a worst-case scenario, the memory requirement becomes similar to that in the page-mapping scheme.

Kang et al. [2006] proposed a superblock-mapping scheme termed "N to N + M mapping." In this scheme, a superblock consists of N adjacent logical blocks, and the superblock is mapped into a group of N + M physical blocks at the page level. M represents the number of the log blocks additionally allocated for the superblock. Normally, N is fixed while M changes dynamically according to the number of currently available log blocks. If a new log block is allocated to the superblock, M is increased by one. Moreover, M is decreased when a merge operation is performed on the log block and the data 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 the amount of extra storage and number of extra flash memory operations, the fine-grain mapping information is stored in the spare area of NAND flash memory. Performance evaluations show that the superblock scheme reduces the level of garbage collection overhead by as much as 40% compared to previous FTL schemes with roughly the same memory overhead. However, this FTL design relies on a limited size of the spare area to maintain the page-mapping table, and the parameters N and M cannot be tailored for the specific requirements of various applications. The proposed technique differs from that of Kang et al.[2006] primarily in that it addresses an efficient design space exploration method for the optimal values of such parameters as N and M when there is no limitations associated with the values of these parameters.

4. THE PROPOSED APPROACH: FLEXIBLE GROUP MAPPING

The proposed flexible group-mapping method is based on the log block scheme. It is similar to the superblock-mapping scheme of Kang et al. [2006].

The basic idea of flexible group mapping is to configure the degree of sharing of log blocks among data blocks using the block-level spatial locality parameter, N, and to manage the degree of the allocation of log blocks for the frequently updated data blocks (known as hot data) using the delayed merge parameter, K. The optimal {N, K} parameters are inferred from the access patterns of the target application as NAND flash applications such as MP3, digital cameras, and PC applications have a tendency to show specific access patterns based on limited user scenarios.

In the flexible-mapping scheme, a data block group is a series of data blocks that consists of N sequential blocks. The parameter N is the number of data blocks in a data block group. The N parameter indicates the associativity among neighboring blocks. It explains the evolution from the log block scheme to FAST in terms of associativity. If N is 1, it corresponds to the log-block scheme; the direct-mapping scheme. If N is the total number of blocks in NAND flash, it corresponds to the FAST scheme[1]—the fully associative mapping scheme.

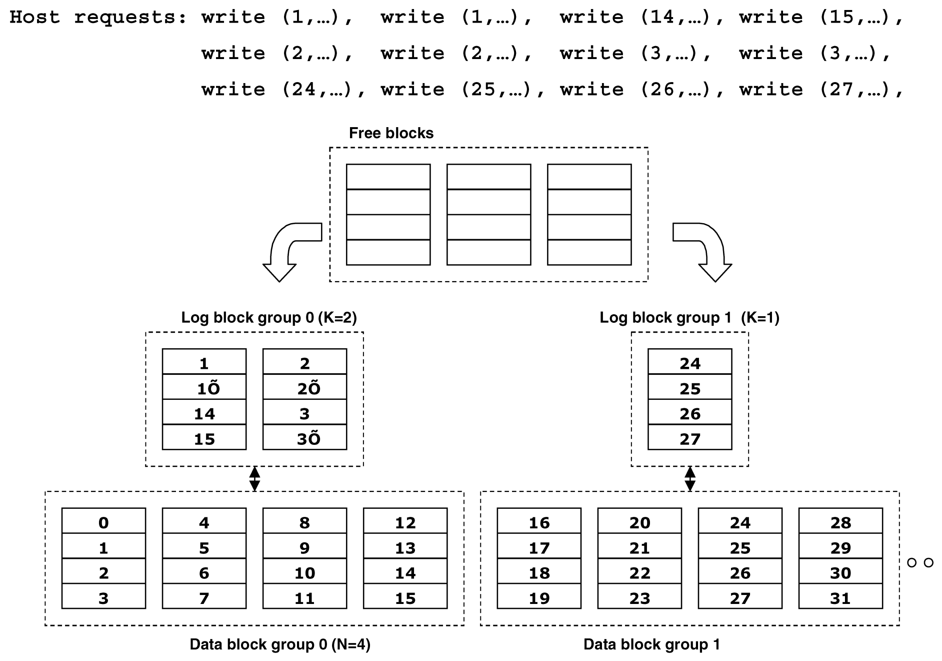
A log block group is a set of log blocks related to a specific data block group. The parameter K denotes the maximum number of log blocks that can be added to a log block group. The K parameter explains the type of temporal locality within a block. If some pages are frequently updated in a block, they are known as hot pages. It is more beneficial to retain hot pages without a merge operation because they are apt to be updated sooner [Chang and Kuo 2002]. In this case, the merge operation of the data block including hot pages is delayed by as much as possible. In other words, the parameter K should be assigned its maximum value. Otherwise, the value of K does not have to be large.

For instance, the scheme in Figure 8 assumes that a series of write requests (1, 1, 14, 15, 2, 2, 3, 3) exists. When the log block group 0 is full of valid and invalid pages, it will be more beneficial to add one more log block than to merge log block group 0 and data block group 0, as the logical pages (1, 2, 3) are likely to be hot pages. On the other hand, when the next write request to a logical address (24, 25, 26, 27) occurs, it is more beneficial to merge log block group 1 into a new data group, as the merge procedure simply requires an update of the mapping table without valid page copy operations. This merge scheme is known as a "switch merge"[2] in Kim et al. [2002].

Although it appears that adding log blocks is always advantageous, this procedure is limited by the number of available free blocks. If some specific data blocks exhaust the log blocks as the K parameter increases, the other data blocks will have to compete with each other for the allocation of log blocks. This may cause unanticipated frequent merge operations. This competition can be lessened by the monopolizing of the volunteer merge operations by data blocks. In the same way, the N parameter contains a trade off between the utilization of log blocks and the increased merge cost (e.g., if N is 4, its merge cost will be increased fourfold compared to when N is 1). As a result, the reasonable determinations of the N and K parameter values are indispensable in the early design stages.

[1]The FAST scheme uses a special block known as a sequential block to efficiently handle long sequential write patterns.

[2]In this article, "switch merge" is referred to as "swap merge" without a loss of generality.



```
Host requests: write (1,…),  write (1,…),  write (14,…), write (15,…),
               write (2,…),  write (2,…),  write (3,…),  write (3,…),
               write (24,…), write (25,…), write (26,…), write (27,…),
```

Fig. 8. Flexible group-mapping scheme.

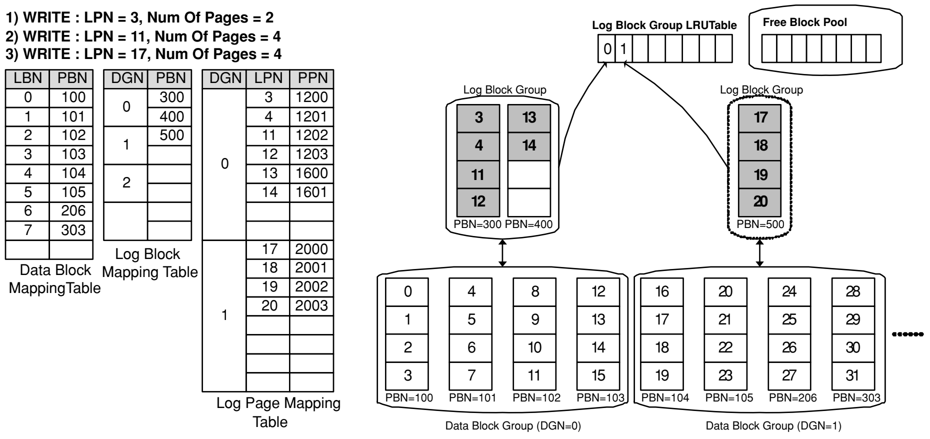
As mentioned earlier, FTL performance is dominated more by the write and merge operations than by read operations. In the following subsections, the write and merge schemes employed in the proposed architecture are presented in detail.

4.1 Write Scheme

In order to translate from a logical to a physical page address, it is necessary to maintain three mapping tables: (1) the data-block-mapping table (DBMT), (2) the log-block-mapping table (LBMT), and (3) the log-page-mapping table (LPMT). The DBMT contains an array of physical block numbers indexed by logical block numbers. The LBMT contains data block group numbers and their associated physical block numbers which are the log block numbers for the data block groups. The LPMT contains associated data block group numbers, logical page numbers, and physical page numbers. In this table, only the page-mapping information of the log block exists. In addition, the bad-block-mapping table (BBMT) is used to replace the initial or runtime bad blocks with reserved blocks.

Figure 9 illustrates the write operation process. In this example, it is assumed that a block consists of four pages (i.e., N is 4 and K is 2). Initially, when a write request is issued, a check is required to determine if the corresponding data block group number (DGN) is associated with a log block group. This is done by searching the LBMT. In this example, the logical page number (LPN) is 3, DGN[3] is 0, and the number of pages to write is 2. If it is assumed that no

[3]DGN = LPN div (N $\times$ the number of pages per block).



LBN	PBN
0	100
1	101
2	102
3	103
4	104
5	105
6	206
7	303

DGN	PBN
0	300
	400
1	500
2	

DGN	LPN	PPN
0	3	1200
	4	1201
	11	1202
	12	1203
	13	1600
	14	1601
1	17	2000
	18	2001
	19	2002
	20	2003

Fig. 9. Write operation in Flexible Group Mapping (N = 4, K = 2).

log block is attached to a data-block group, a new log block must be allocated for the data block group from the free block pool. Consequently, the physical block number (PBN) 300 is written in the LBMT, and two pages (LPN: 3, 4) are then written in the PBN block 300.

The LPN and associated physical page number (PPN) are written in the LPMT as a result of the page-mapping operation. When the second write request comes (LPN = 11, the number of pages = 4), the corresponding DGN 0 exists in LBMT, but there are not enough empty pages in the log block (PBN = 300) to write four pages. For this reason, it is necessary to allocate a new log block from the free block pool. If it succeeds in obtaining a free block, the allocated PBN is written in LBMT as (DGN = 0, PBN = 400). Following this, four pages (LPN: 11, 12, 13, 14) are written in the PBN blocks 300 and 400. In addition, the LPNs and PPNs (1202, 1203, 1600, 1601) are written in LPMT. When the third write request comes (LPN = 17, the number of pages = 4), the corresponding DGN 1 is not found in LBMT and it is necessary to create a new log block group. If it succeeds in gaining a free block, the allocated PBN is written in LBMT (with DGN = 1, PBN = 500). Four pages (LPN: 17, 18, 19, 20) are then written in PBN block 500. In addition, the LPN and PPNs (2000, 2001, 2002, 2003) are written in the LPMT. If there are no free blocks in the free block pool, it is necessary to reclaim free blocks using a merge operation.

4.2 Merge Schemes

Figure 10 shows an example of a simple merge operation when *N* is 4 and *K* is 2. Before processing the merge operation, it is necessary to determine the log block that will serve as the merge target. In the proposed scheme, an LRU (least recently used) policy is adopted as this policy is considered to be one of the best replacement policies [Hennessy and Patterson 2003]. According to the LRU policy, a log block can be selected as a merge target. In this example, it is assumed that the log block (PBN = 300) is selected as a merge target. As the log block (PBN = 300) has four valid pages from different data blocks, the merge operation requires 16 page read/program operations and five block erase

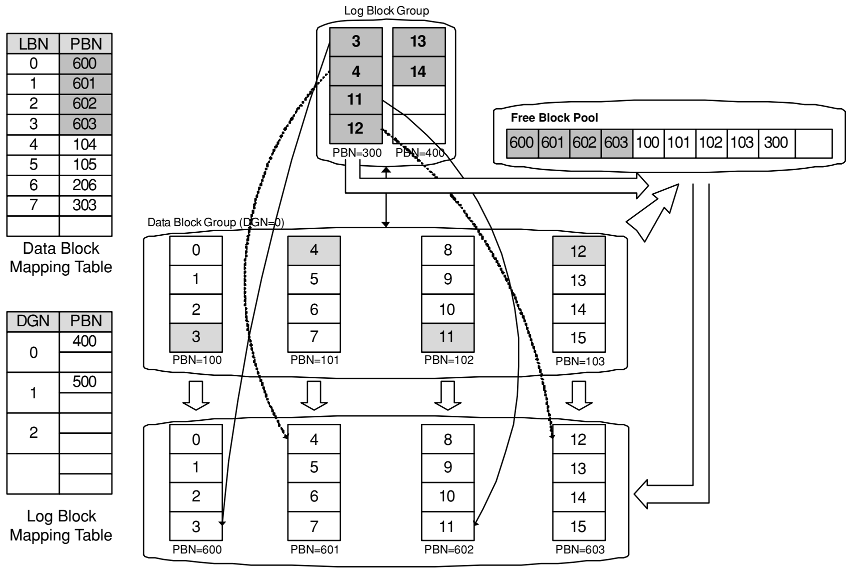


Fig. 10. An example of a simple merge operation (N = 4, K = 2).

operations (one for the log block and four for the data blocks). Each data block in the data block group should be newly allocated, and valid pages are copied into the new data block. The old data blocks (PBN = 100, 101, 102, 103) and log block (PBN = 300) are then inserted into the free block pool. Only when a merge operation occurs does the DBMT change. After the merge operation, a newly allocated data block number is written in the DBMT with (LBN: PBN) = (0:600, 1:601, 2:602, 3:603). In addition, the log block entry (DGN = 0:PBN = 300) is removed from the LBMT (in a comparison to the LBMT in Figure 9).

Figure 11 shows an example of a swap-merge operation. If the log block is occupied by in-place valid pages, reclaiming of the free block can be done by the swap merge. In this example, the log block (PBN = 300) is written to the same pages of the data block (PBN = 101). Thus, the log block becomes a new data block and the old data block (PBN = 101) is inserted into the free block pool. The DBMT is updated with (LBN:PBN) = (1:300).

Finally, Figure 12 shows an example of a copy-merge operation. If the log block is partially occupied by in-place pages of a data block, a free-block reclaim operation can be done by the copy merge. In this example, the log block (PBN = 300) becomes a new data block by copying two pages from the old data block (PBN = 101), and the old data block (PBN = 101) is inserted into the free block pool. The DBMT is updated with (1:300).

5. PERFORMANCE MODEL AND ANALYSIS

Here, flexible FTL architecture that considers not only spatial locality through the associativity parameter N, but also temporal locality through the delayed merge parameter K is presented. Though these parameters effectively

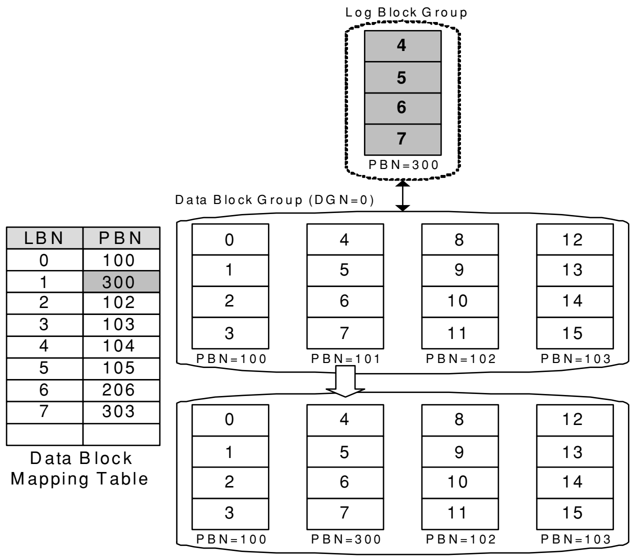


Fig. 11. An example of a swap merge operation (N = 4, K = 2).

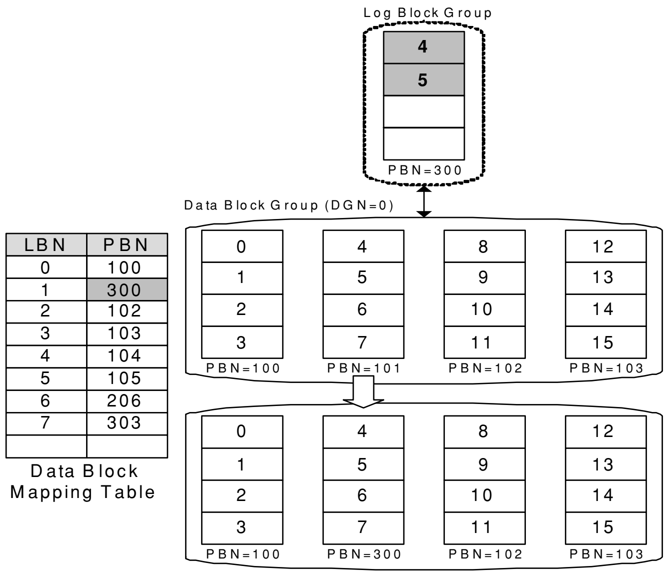


Fig. 12. An example of a copy merge operation (N = 4, K = 2).

configure the FTL architecture and can be geared toward specific NAND flash applications, finding an optimal parameter set of $\{N, K\}$ will require a great amount of time if an efficient design space-pruning method is not provided. For example, an exhaustive simulation method is not preferable as this requires that the exploration cost is multiplied by the number of $\{N, K\}$ combinations, the length of a given trace set, and the FTL execution time. In this section, an efficient exploration method and performance model for exploring the design space of $\{N, K\}$ over a given workload is described.

5.1 Workload Analysis

The storage access patterns are investigated in terms of the request density. Here, $R = \langle R_0, R_1, \ldots, R_{M-1}\rangle$ is a sequence of write requests in a given workload and R_k denotes the kth write request $(0 \leq k < M)$. The entire group of write requests are divided into a series of nonoverlapping request windows W_j of size $|W|$, which contains $\langle R_{j\cdot|W|}, \ldots, R_{(j+1)\cdot|W|-1}\rangle$ for $j = 0, 1, \ldots, N_W - 1$, where N_W represents the total number of request windows.[4]

The request density, $RD_{i,j}$, is defined as the ratio of the number of requests accessed in the ith logical block ($C_{i,j}$) to the total number of requests in the jth window such that $RD_{i,j} = C_{i,j}/|W|$. From this definition, it is apparent that the following equation holds:

$$\sum_{i \in \text{all LBNs}} RD_{i,j} = \frac{1}{|W|} \sum_{i \in \text{all LBNs}} C_{i,j} = 1 \quad \text{for any request window} W_j.$$

Here, an example sequence of write requests, $R = \langle R_0, R_1, \ldots, R_{19}\rangle$, issued to data blocks from LBN0 to LBN3, is considered, as shown in Figure 13(a). In Figure 13, it is assumed that each data block consists of four pages, i.e., $|W| = 4$. Figure 13(b) illustrates the corresponding request density table in which $RD_{i,j}$ is shown for each LBN i and the request window W_j. It is important to note that because LBN2 receives one write request out of the total of four requests during W_3, $RD_{2,3}$ is calculated as 0.25 in the request density table.

Each $RD_{i,j}$ for W_j can be used as a clue to determine the most appropriate value of N. The associativity parameter N determines how many LBNs are to be assigned to one log block group. If the request density is high, a small number of LBNs may suffice to capture the spatial locality. For example, the log block group associated with $RD_{2,3}$ may require more log blocks than $RD_{0,4}$, as LBN2 alone does not have adequate spatial locality in W_3. In fact, the number of LBNs to be allocated to a single log block group can be obtained by taking the inverse of the request density; of concern here is the minimum value N_j for a given window W_j such that

$$N_j = \left\lfloor \min\left(\frac{1}{RD_{i,j}}\right)\right\rfloor \quad \text{for } i \in \text{ all LBNs.}$$

N_j is regarded as a candidate value for N in the request window W_j, and values of N_j obtained for the example trace are displayed at the bottom of the request distribution table in Figure 13(b). If N is large, high utilization can be attained for one log block group because many LBNs that exhibit a low request density share the same log block group. However, this may lead to high merge costs when a log block is reclaimed, which can degrade the performance. Thus, it is necessary to determine the minimum N value while obtaining a utilization

[4]Unless otherwise stated explicitly, it is assumed that $|W|$ equals the number of pages inside a block in flash memory. Hence, $|W| = 64$ for typical NAND flash memory.

Requests (LPN) : 0, 1, 2, 3, 4, 6, 4, 6, 8, 9, 13, 15, 13, 15, 10, 12, 3, 3, 10, 13

		W_0	W_1	W_2	W_3	W_4
LBN 0	Page 0	R_0				
	Page 1	R_1				
	Page 2	R_2				
	Page 3	R_3				R_{16}, R_{17}
LBN 1	Page 4		R_4, R_6			
	Page 5					
	Page 6		R_5, R_7			
	Page 7					
LBN 2	Page 8			R_8		
	Page 9			R_9		
	Page 10				R_{14}	R_{18}
	Page 11					
LBN 3	Page 12				R_{15}	
	Page 13			R_{10}	R_{12}	R_{19}
	Page 14					
	Page 15			R_{11}	R_{13}	

(a) An example request distribution table

	W_0	W_1	W_2	W_3	W_4
LBN0	1.00	0.00	0.00	0.00	0.50
LBN1	0.00	1.00	0.00	0.00	0.00
LBN2	0.00	0.00	0.50	0.25	0.25
LBN3	0.00	0.00	0.50	0.75	0.25
N_j	1	1	2	4	4

(b) Estimating N_j

	W_0	W_1	W_2	W_3	W_4	K
LBN0	0	0	0	0	1	1
LBN1	0	1	1	1	1	1
LBN2	0	0	0	0	1	1
LBN3	0	0	0	1	2	2

(c) Estimating K_i

Fig. 13. Estimating N_j and K_i from an example trace.

rate that is as high as possible. One means of determining the optimal value of N is to construct histogram of N_j values and evaluate only the several topmost values of N_j as the possible candidates for the optimal N. Section 6 shows that this methodology is reasonably effective in reducing the exploration space.

On the other hand, the optimal value of the delayed merge parameter K can be predicted by measuring the temporal locality in each LBN. Essentially, it is considered that LBN i has temporal locality in the request window W_j if one or more pages in LBN i are updated more than once during W_j or if one or more pages in LBN i written in the previous window W_{j-1} are written again in the current window W_j. Here, for example, $K_{i,j}$ is considered as the number of occurrences that satisfies such conditions during the interval from W_0 to W_j for LBN i. When $PAGE_{i,j}$ denotes a set of pages in LBN i written during request window W_j, $K_{i,j}$ can be calculated by the following recurrence relationship:

$$\begin{aligned} K_{i,0} &= 0 \\ K_{i,j} &= K_{i,j-1} + d_{i,j} \quad \mathit{for}\ j > 0 \\ \mathit{where}\ \ d_{i,j} &= \begin{cases} 1\ \mathit{if}\ \ |PAGE_{i,j}| < C_{i,j} \vee \left(\sum_{k=0..j-1} PAGE_{i,k} \cap PAGE_{i,j} \right) \neq \Phi. \\ 0\ \ \mathit{otherwise}. \end{cases} \end{aligned}$$

Figure 13(c) presents the values of $K_{i,j}$. In Figure 13(c), $K_{0,4}$ and $K_{1,1}$ have been increased by one as two pages are updated in the same window, while $K_{2,4}$,

$K_{3,3}$, and $K_{3,4}$ have been increased as the pages written in the previous window are updated again. Here, $K_{i,j}$ does not increase when $|PAGE_{i,j}| = |W|$, as in this case it is possible to perform a swap-merge operation without requiring an additional log block.

In an ideal situation where the number of log blocks is sufficient, the final value of $K_{i,j}$'s, namely $K_i = K_{i,N_W-1}$, represents an update frequency that is directly related to the number of log blocks that should be given to the particular value of LBN i. A larger value of K_i implies that more pages are updated in the logical block, thus requiring more log blocks. As with the associative parameter N, the optimal value of K for all LBNs can be found by evaluating only statistically significant values among K_i.

5.2 Performance Analysis

For the simplicity of analysis, it was assumed that the size of every write request is only one sector. Given that any write request with more than one sector can be converted into a series of one sector write request, this is a reasonable assumption.

Here, A_k is the active data block accessed by R_k and AG_k is the active-data block group accessed by R_k. Hence, AG_k always contains A_k from the definition. In addition, $SA(W_j)$ and $SAG(W_j)$ is the set of active blocks and the set of active block groups that are accessed by the requests issued in W_j, respectively. Finally, the number of log blocks associated with AG_k is defined as $L(AG_k)$.

When request R_k arrives in request window W_j, a merge operation occurs if the following condition (Eq. 1) is satisfied, where LB is the maximum number of log blocks available in the system.

$$\sum_{AG_k \in SAG(W_j)} L(AG_k) > LB \tag{1}$$

From the definition of K, the number of log blocks for each active group is smaller than K; that is, $L(AG_k) \leq K$. Thus, Eq. (1) can be converted as follows:

$$|SAG(W_j)| \times K^{\partial} > LB \quad \text{for } \partial(0 \leq \partial \leq 1) \tag{2}$$

The constant ∂ is associated with the update frequency for the input pattern. If a large portion of requests in W_j access an active group, the group may have more log blocks than any other active groups. In such a case, the value of ∂ may be close to 0 because K is not a strict condition that leads to a merge operation. On the other hand, if the requests in W_j are evenly scattered over many active groups, the requests attempt to obtain their log blocks competitively, which results in many merge operations. In this case, $|SAG(W_j)|$ is larger than that of the first case, and the large value of K can invoke more frequent merge operations. Thus, the value of ∂ is close to 1.

Moreover, the value of $|SAG(W_j)|$ can be expressed by $\frac{|SA(W_j)|}{N^{\varepsilon}}$ for $0 \leq \varepsilon \leq 1$. If the size of W_j is reasonably large, and the request types of the input pattern do not have much variation, $SA(W_j)$ can be approximately expressed by a constant C. The value of ε associated with the associativity of the pattern. If the associativity is strong, the active blocks may be consecutive data blocks.

Thus, the number of active group becomes similar to that of the active blocks divided by N ($|SAG(W_j)| \approx \frac{|SA(W_j)|}{N}$), as a block group has N consecutive data blocks. From this result, the value of ε can go to 1. On the other hand, if the input pattern has poor associativity, the value of ε becomes nearly 0. From these results, the equation below can be obtained.

$$C \times \frac{K^{\partial}}{N^{\varepsilon}} > LB \quad \text{for a constant } C. \tag{3}$$

From this equation, an analysis of the performance of the MP3 and PC patterns can be performed. In the MP3 pattern, the requests in a window access a very small number of blocks. Thus, ∂ is close to 0. Moreover, it has poor associativity in a request window and N is also close to 0. In this case, the sizes of N and K are not major factors in the creation of a merge operation. From Figure 19, it is known that N and K do not affect the performance strongly.

In the access pattern of a PC application, the requests in a request window access many active log groups evenly and the value of ∂ is close to 1. In addition, it has more associativity than the MP3 pattern. However, the associativity is not strong and ε is not large. In this case, a large value of K can result in many merge operations and poor performance can result. In addition, N does not have significant effects upon the write performance. From Figure 18, it is clear that a small value of K produces superior performance.

5.3 Memory Requirement Analysis

The memory usage for the active log group was computed. An active log group has three data structures. First, it maintains a page map table for $64N$ pages in the log group. As the memory usage for the information for each page is constant, the total memory usage for the table is c_1N bytes. Next, the active log group maintains some information for one or more log objects. A data structure related to a log block has a small constant number of variables that approximates the number of valid pages in the log. The memory for this information is, at most, c_2K because the log group has K logs. Finally, log group has a small number of additional variables that approximates the number of logs. From these results, the total memory requirement can be expressed by the following equation:

$$\textit{Memory requirement} = |SAG(W_j)| \times (c_0 + c_1N + c_2K)$$

6. EXPERIMENTAL RESULTS

The traces analyzed in this article were collected using an Intel Pentium-4 PC system with 512 MB of RAM and an 80 GB hard disk. The operating system was Windows XP, and the file system was NTFS. The traces were obtained from an in-house monitoring tool for a disk-access pattern.

Upon closer inspection of the trace data and the statistical analysis of "Internet/MS Office use case" in Figure 14, significant randomness in storage accesses are found, as small temporary files are created and deleted during Internet surfing activities. In addition, accesses related to the internal

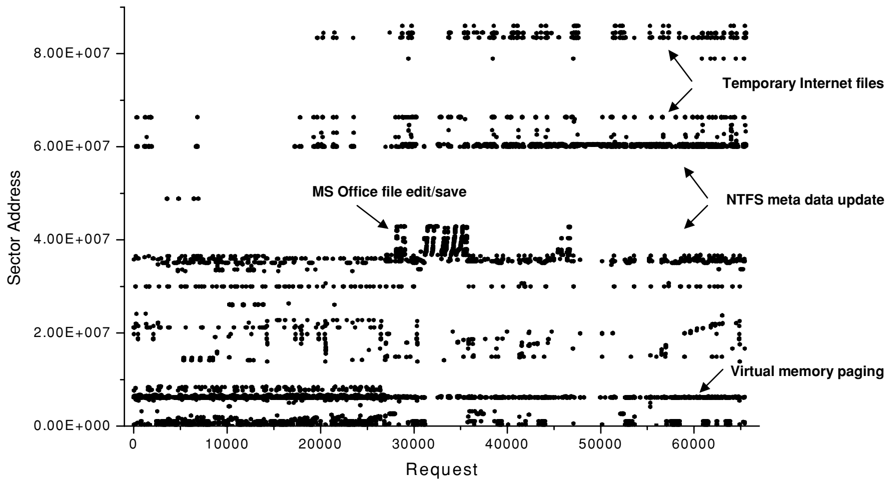


Fig. 14. Trace distribution from PC applications.

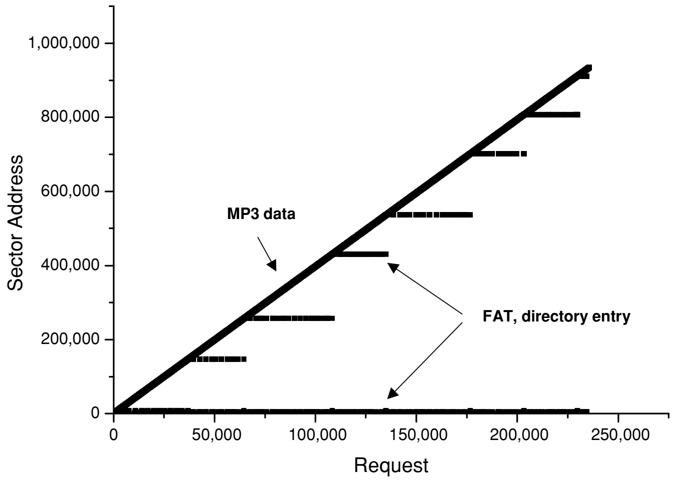


Fig. 15. Trace distribution from an MP3 download.

activities of the operating system were found, which included virtual memory paging and metadata updates of the file system. These simultaneous read/write requests to storage devices are multiplexed in storage systems. This observation implies that multiple working sets exist at the same time. For example, “Internet Explorer,” “MS Office,” “Virtual memory manager,” and “File System” are independently accessing the storage system with their own strong spatial locality and high tendencies to update their data with temporal locality.

On the other hand, the “MP3 file download use case” exhibits a mostly sequential access pattern, even though there are small-sized random requests because of file system metadata (e.g., FAT, directory entry) updates (see Figure 15).

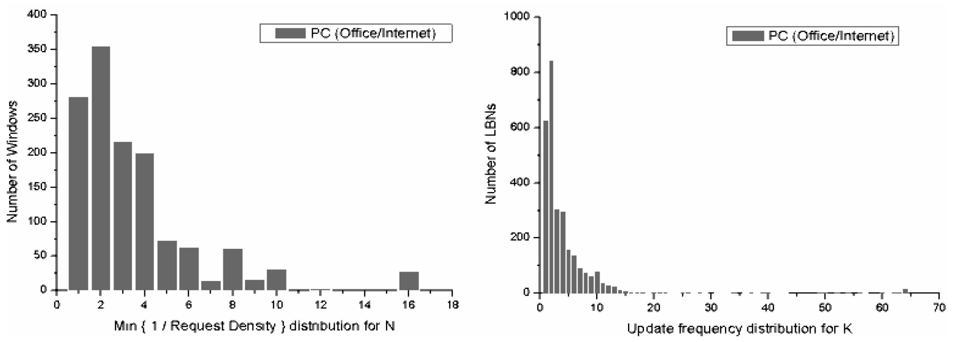


Fig. 16. Distribution of N's and K's for PC applications.

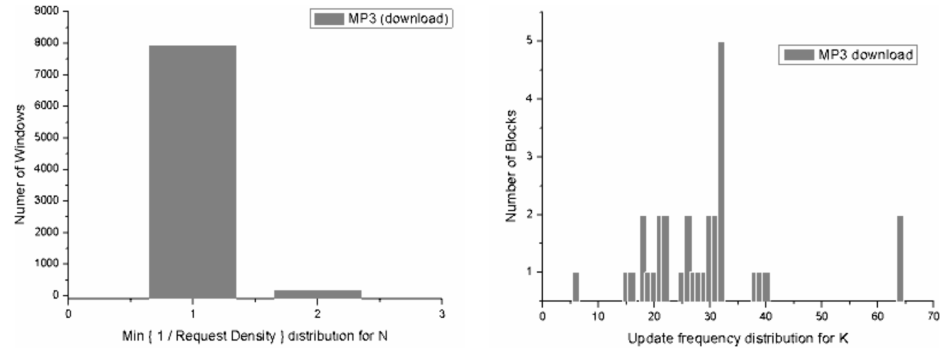


Fig. 17. Distribution of N and K values for the MP3 application.

From the storage access patterns shown in Figures 14 and 15, the following distributions are obtained for possible candidates of N and K. Each distribution is obtained as described in Section 5.1. The result of $\lfloor \min(\frac{1}{RD_{i,j}}) \rfloor$ is calculated for each window W_j and the distribution of the values are obtained, as shown in Figure 16 (left). The number of updates within each LBN are also counted and the distribution of K obtained, as shown in Figure 16 (right).

In the experiments, the PC applications show more associativity than the MP3 case, as MP3 file downloading results in a sequential write pattern. In Figures 16 and 17, the values of $2 \sim 8$ for the PC, and $1 \sim 2$ for the MP3 can be taken as possible candidates for the parameter N by considering the deviation.

Similarly, the values of $4 \sim 8$ for PC and $10 \sim 30$ for MP3 can be selected as the possible candidates for the parameter K. Particularly, sequential requests have little effect on the performance according to N and K. Accordingly, the distributions for other types of requests apart from sequential requests is sparse and contains a large amount of deviation.

In order to verify the usefulness of the proposed design space-pruning method, all the pairs of N and K were simulated and a performance map was created. The map shows performance of the every combination of $\{N, K\}$, from the best case to various degraded cases. As shown in Figure 18, the

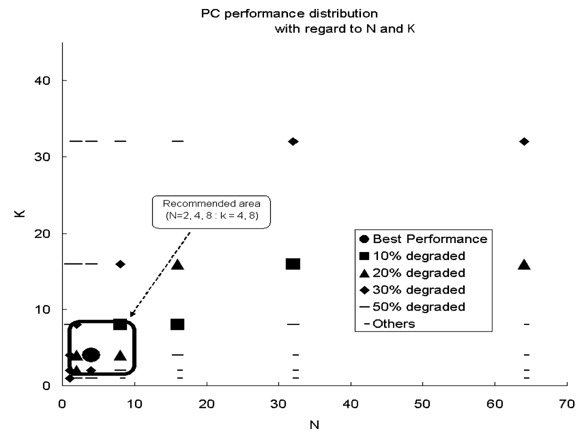


Fig. 18. PC application performance variation with the change of N and K.

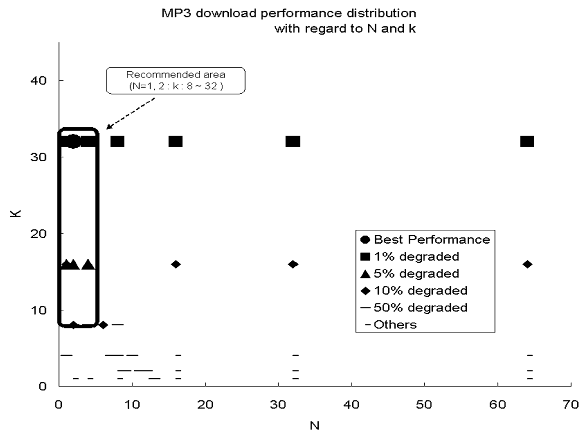


Fig. 19. MP3 download performance variation with the change of N and K.

recommended sets of $\{N = 2, 4, 8, K = 4, 8\}$ include the best performance combination of $\{N, K\}$, as expected.

With the MP3 usage case, the recommended sets of $\{N = 1, 2, K = 8{\sim}32\}$ include the best performance combination of $\{N, K\}$, as expected (cf. Figure 19).

7. CONCLUSIONS

This article introduced a reconfigurable FTL architecture to efficiently handle diverse NAND flash applications ranging from MP3 to SSD for a PC. The associativity between data blocks was parameterized using N, the number of data blocks in a data block group, and K, the maximum number of log blocks in a log block group that belong to a group of N data blocks. In order to efficiently explore the design space, a workload analysis method based on the density distribution of given requests and the update frequency is proposed. The experimental results show that the proposed architecture can be reconfigured to a given workload ranging from MP3 to PC applications and that the proposed analysis method can efficiently find the optimal N and K values within a reasonable amount of time.

REFERENCES

BAN, A. 1995. Flash file system. United States Patent, No. 5,404,485 (Apr.).

CHANG, L. P. AND KUO, T. W. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*.

CHANG, L. P. AND KUO, T. W. 2004. An efficient management scheme for large scale flash memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, New York. 862–868.

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

GAL, E. AND TOLEDO, S. 2005. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37*, 138–163.

HENNESSY, J. L., AND PATTERSON, D. A. 2003. *Computer Architecture: A Quantitative Approach* 3rd Ed. Morgan Kaufmann, Burlington, MA.

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 ACM/IEEE Conference on Embedded Software (EMSOFT'06)*. Seoul, S. Korea.

KIM, J. S., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. K. 2002. A space-efficient flash translation layer for compact flash systems. *IEEE Trans. Cons. Elect. 48*, 366–375.

LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG H.-J. 2006. A log buffer based flash translation layer using fully associative sector translation. *ACM Trans. Embed. Comput. Syst.*

MIN, S. L. 2004. Love/hate relationship between flash memory and microdrive for low-power portable storage. In *1st International Workshop on Power-Aware Real-Time Computing*, Pisa, Italy.

SAMSUNG ELECTRONICS. 2005. NAND Flash Memory & Smart-Media Data Book.

Received September 2007; accepted December 2007