Block Recycling Schemes and Their Cost-based Optimization in NAND Flash Memory Based Storage System

Jongmin Lee
School of Computer Science
University of Seoul
Seoul, Korea
jmlee@uos.ac.kr

Sunghoon Kim
Center for the Info. Security Tech.
Korea University
Seoul, Korea
kimsunghoon@korea.ac.kr

Hunki Kwon
School of Computer Science
University of Seoul
Seoul, Korea
kwonhunki@uos.ac.kr

Choulseung Hyun
School of Computer Science
University of Seoul
Seoul, Korea
cshyun@venus.uos.ac.kr

Seongjun Ahn
Software Laboratories
Samsung Electronics Co.
Seoul, Korea
Seongjunahn@gmail.com

Jongmoo Choi
Division of Information and CS
Dankook University
Seoul, Korea
choijm@dankook.ac.kr

Donghee Lee*
School of Computer Science
University of Seoul
Seoul, Korea
dhl_express@uos.ac.kr

Sam H. Noh
School of Computer and Information Engineering
Hongik University
Seoul, Korea
Samhnoh@hongik.ac.kr

ABSTRACT

Flash memory has many merits such as light weight, shock resistance, and low power consumption, but also has limitations like the erase-before-write property. To overcome such limitations and to use it efficiently as storage media in mobile systems, Flash memory based storage systems require special address mapping software called the FTL (Flash-memory Translation Layer). Like *cleaning* in Log-structured file system (LFS), the FTL often performs a *merge* operation for block recycling and its efficiency affects the performance of the storage system. To reduce the block recycling costs in NAND Flash memory based storage, we introduce another block recycling scheme that we call *migration*. Our cost-models and experimental results show that cost-based selection of merge or migration for each block recycling can decrease block recycling costs and, therefore, improve performance of Flash memory based storage systems. Also, we derive the macroscopic optimal migration/merge sequence minimizing block recycling costs for each migration/merge combination period. Experimental results show that the performance of Flash memory based storage can be further improved by the macroscopic optimization than the simple cost-based selection.

Categories and Subject Descriptors

C.5.3 [**Microcomputers**]: Portable Devices; D.4.2 [**Operating System**]: Storage Management—Secondary storage

General Terms

Design, Performance, Experimentation

Keywords

Flash memory based storage system, merge operation, migration operation, FTL (Flash-memory Translation Layer)

1. INTRODUCTION

Flash memory has been used as storage media in mobile devices such as cellular phones and PDAs and is now to be used in personal computers in the name of solid state disks (SSD). Flash memory has advantages over conventional magnetic disks in terms of weight, shock resistance, and power consumption. However, it also has some inherent limitations such as the erase-before-write requirement and different unit sizes for read/write and erase operations. Therefore, special address mapping software called the Flash-memory Translation Layer (FTL) was introduced into Flash memory based storage systems to overcome these limitations [6]. The FTL is a software layer that emulates an array of logical sectors

* Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EMSOFT'07, September 30-October 3, 2007, Salzburg, Austria.

Copyright 2007 ACM 978-1-59593-825-1/07/0009...$5.00.

over the Flash memory media by translating a logical sector address to a physical data location on Flash memory.

When a read request is given with a sector address, the FTL searches a map to find the location of the sector data on Flash memory and, if the search is successful, it reads and delivers the sector data. In case of a write request, the FTL allocates a new empty space for that sector and writes the sector data on it. It then modifies the mapping information for the sector. Overall, the operations of the FTL is similar to Log-structured file system (LFS) [16] in that it remaps the sector location to a new area for each write request and space need to be reclaimed via a space recycling scheme [13]. Specifically, in NAND Flash memory, erase units called *blocks* need to be recycled and, henceforth, we call the space recycling scheme in NAND Flash memory FTL as a *block recycling scheme*. This block recycling scheme in NAND Flash memory FTL is also often called a *merge* operation because it collects sectors scattered on two or more blocks into a block and the other blocks are reclaimed. In this paper, we assume that the FTL uses the merge operation for its recycling scheme.

Like the LFS where its performance depends on the efficiency of the space recycling scheme, the block recycling efficiency of the FTL, specifically the merge efficiency, affects the performance of Flash memory based storage systems. Moreover, its efficiency varies according to the data write pattern as can be similarly observed with the LFS. If data were written sequentially, the merge operation can reclaim a whole block with low cost and, conversely, for randomly or repeatedly written data the block reclaiming cost is quite expensive. Unfortunately, we can observe both sequential and repeated write patterns in file system requests and, thus, we need to find an efficient block recycling scheme for repeated write as well as for sequential write.

To improve block recycling efficiency for the repeated write pattern, we introduce a new block recycling scheme, which we call the *migration* operation. Specifically, the migration operation can reduce the block recycling costs for repeated write patterns that forces the costly merge operation to reclaim a block. Also, we present cost models of the merge and the migration operations that enable the FTL to select a low cost operation between them when it comes to recycle a block. Experiments with the Postmark benchmark and embedded system workloads show that the cost-based selection of migration or merge operation reduces block recycling costs of FTL and, as a result, improves the performance of Flash memory based storage. During the experiments, we found the existence of a macroscopic optimal migration/merge sequence that minimizes block recycling costs for each migration/merge combination period. Experimental results of the benchmark and the workloads show that the performance of Flash memory based storage can be further improved with the macroscopic optimization than the simple cost-based selection.

This paper is organized as follows. In the next section, we describe characteristics of Flash memory and related works. In Section 3, we provide basic knowledge regarding the block recycling scheme, specifically the merge operation in FTL. In Section 4, we introduce the migration operation and then derive the cost models of the migration and merge operations. In Section 5, we present a macroscopic optimization that minimizes block recycling costs for each migration/merge combination period. Section 6 presents the experimental results of the cost-based selection and the macroscopic optimization, and finally we conclude this paper in Section 7.

2. FLASH MEMORY CHARACTERISTICS AND RELATED WORKS

NAND Flash memory has unique characteristics as follows. NAND Flash memory consists of the same size blocks, each of which in turn consists of the same size pages. The block size and the page size are typically 16 to 256 KB and 0.5 to 4 KB, respectively. Read/write data from/to NAND Flash memory is performed in page units. SLC (Single Level Cell) Flash memory typically takes 25 µs for page read and 300 µs for page write, respectively, while for MLC (Multi Level Cell) Flash memory it takes considerably longer for both page read and page write operations. Data in Flash memory, once written to a page, cannot be updated in place. This means that to update a page without an FTL the block containing the page must be erased and that all pages in the block must be written again along with the new data. Such an erase operation is performed in block units and typically takes 2 ms in SLC Flash memory and 1.5 ms in MLC Flash memory, respectively [1, 2].

According to their mapping unit, typical FTLs can be divided into two categories, a page mapping FTL and a block mapping FTL. A page mapping FTL calculates a logical page number with a requested sector number and then translates the logical page number to a physical location (a page in a block) through a map. In case of block mapping FTL, a physical block has a fixed number of sectors in ordered manner on its pages. When a sector read/write request is given, a block mapping FTL calculates a logical block number by dividing the requested sector number with the number of sectors in a block and then translates the logical block number to a physical block number through a map [4, 10, 14]. Then, the FTL can easily find a sector location within the physical block because sectors are stored at that block typically in ascending order.

Page mapping FTLs have been used in low-capacity NOR Flash memory cards while block mapping FTLs have been used in most high-capacity NAND Flash memory cards [3, 5] because the map size for block mapping is much smaller than for page mapping. As mentioned earlier, performance of a page mapping FTL depends on space recycling efficiency just as the performance of LFS depends on *cleaning* efficiency [13]. Also, as performances of LFS and LFS-like Flash memory file systems such as JFFS [17] and YAFFS [7] have been improved by separating hot and cold data, space recycling efficiency of a page mapping FTL could be improved by separating hot and cold data [8, 9].

Block mapping FTLs have some advantages in that the map size is small and overall operation is simple. Also, HW/SW co-design of frequent operations can boost the performance of an FTL as we can see in solid state disks. However, the performance of a block mapping FTL is degraded in case of random and repeated write patterns. To overcome this problem, Kang et al. proposed a hybrid scheme in which a mapping unit is a super block (a group of logically adjacent blocks) while page mapping is used inside a super block [12]. Also, Lee et al. proposed another hybrid approach called FAST (Fully Associative Sector Translation) [15], where the sector mapping (similar to page mapping) was introduced restrictively to the write buffer area (Log blocks described in Section 3) while the mapping unit for the data area is still in blocks. Experimental results of the FAST scheme show that the complementary sector mapping is helpful in improving the performance of Flash memory storage, specifically, in case of random and repeated write patterns. However, these hybrid schemes require additional implementations of page mapping features and garbage collection mechanisms for

page mapping areas aside from the original block mapping FTL. In contrast, the migration operation proposed in this paper can be implemented with minor modifications of the original merge operation, but the improvement in performance is limited to repeated write patterns only. Eventually, the FTL designers will have to choose between the hybrid schemes or our approach to improve performance of non-sequential write patterns based on the given resources and performance requirements,

3. FLASH-MEMORY TRANSLATION LAYER AND *MERGE* OPERATION

As mentioned earlier, an FTL makes Flash memory appear to the upper file system layer like a virtual block device. For this purpose, the FTL creates virtual sectors on Flash memory and manages them so that they seem to be updated in place. Also, the FTL reclaims blocks with a block recycling scheme when it needs free space for virtual sector updates. Before we introduce the new block recycling scheme that we call *migration*, we will explain an existing bock recycling scheme called the *merge* operation. In explaining these FTL operations, we assume that the FTL uses a block mapping scheme in translating the logical address to a physical address. Though the commercial FTL used in our experiments has sophisticated features such as multi-level mapping, wear-leveling, and other performance and reliability enhancement schemes, those features do not conflict with the fundamentals of the block recycling schemes and the cost functions described in this paper.

As existing data in a Flash memory block cannot be updated in place, FTL redirects a sector write request to an unused page in a temporary block which we call henceforth a *log block*. In Fig. 1(b), repeated write requests to sectors 8 and 9 are redirected to pages in a log block. Subsequent write requests will eventually consume all pages in the log block. Then, a merge operation will need to be performed to reclaim a log block to service subsequent write requests. A merge operation is composed of erasing a new empty log block and copying all valid sectors from either the old log block or the old data block to the new empty log block (Fig. 1(c)). In the example in Fig. 1(c), after copying all sectors, the new log block becomes the data block for sectors 8~11 and both the old data block and the old log block become free log blocks that will be used for subsequent write requests or later merge operations. Last of all, the relevant map is updated to reflect the new status of the data and log blocks.

If all sectors 8~11 are written sequentially in the log block, an optimized form of the merge operation, sometimes called the *switch merge*, is possible. For switch merge, the FTL, without erasing a new log block and the copying of pages, simply updates the map to designate the log block as the new data block for sectors 8~11, and the old data block for those sectors is designated a free log block. With this optimization, block recycling costs for sequential write patterns become much lower than those for random or repeated write patterns that require the full merge procedure of erasing a block and copying all pages as depicted in Fig. 1.

By the way, we can easily observe not only sequential write patterns, but also repeated such patterns from the upper layer file system. In Fig. 2, we see a mixture of two types of write requests, sequential write of file data and repeated update of metadata such as FAT and directories. We think that these mixed write patterns are common in many file systems and that an optimization is also needed for this repeated write besides the already efficient sequential write.

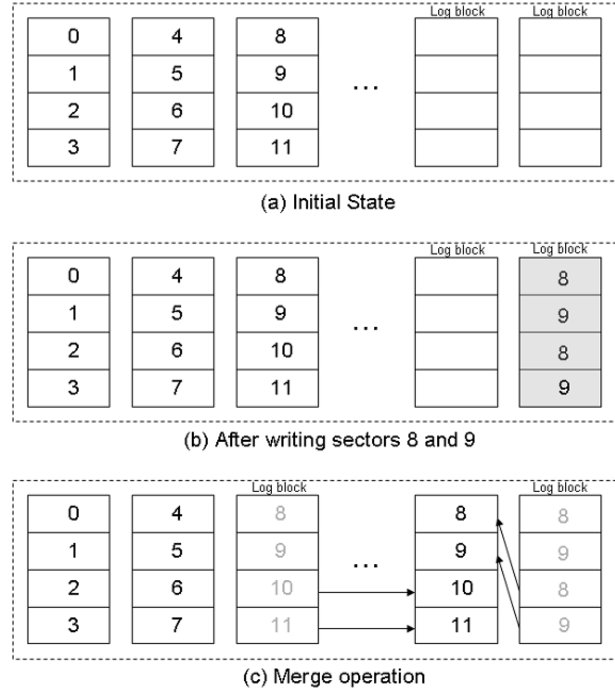

Figure 1. Merge operation in FTL

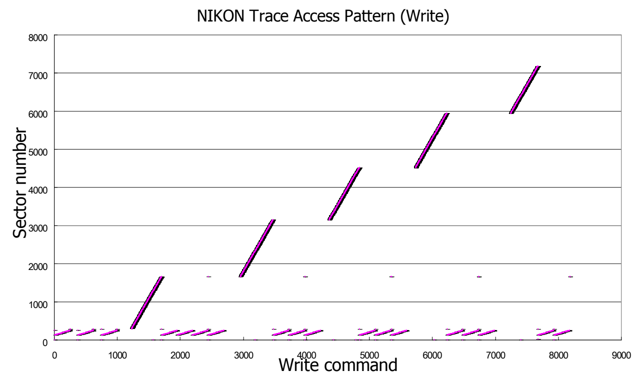

Figure 2. Write pattern of a FAT file system running on Flash memory based storage of a Nikon camera

4. MIGRATION OPERATION AND COST MODELS

This section describes the new block recycling scheme for repeated write patterns that we refer to as the *migration* operation. Also, we derive the cost models for the merge and migration operations that are used by the FTL to select the more cost effective method when reclaiming a block.

If writes to some sectors are repeatedly requested, then those sectors will be redirected to a log block until all pages in the log block are consumed. When all the pages have been used, as we know, a merge operation must be performed to generate a free block and, after the merge, all pages in the new free block can be used again

for subsequent write requests. Observe in Fig. 3(b), that only two pages are written repeatedly in the log block. When a relatively small number of pages are written repeatedly in a log bock, we have an opportunity to apply a *migration* operation and eventually, to reduce block recycling costs. Unlike the merge operation that erases a new log block and copies all pages from the old data block and/or the old log block, the migration operation erases a new log block and copies only the valid pages, two pages in Fig. 3(b), from the old log block to the new log block. The old log block, then, becomes a free block, while the new log block becomes a write buffering area for sectors 8~11 to which all incoming write requests for those sectors will be redirected.

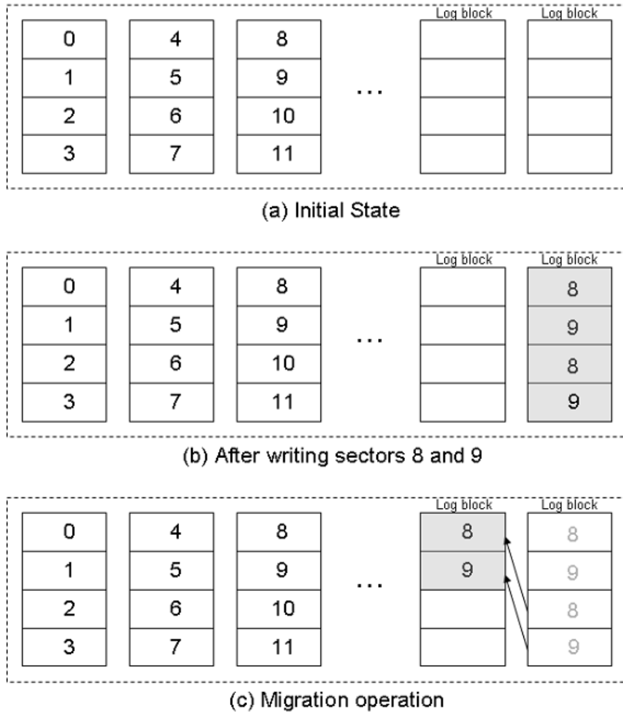

Figure 3. Migration operation

Assume that the number of pages in a block is N_p and the number of valid unique pages in the old log block is p (p=2 in Fig. 3(b)). Then, the migration operation copies p pages from the old log block to a new log block and, as a result, N_p-p pages are available in the new log block after migration. Generally, the merge operation pays a high price, but produces all N_p pages available in the new log block, while the migration operation pays a lower price, but produces a relatively small number of pages that are available in the new log block. Therefore, a cost-benefit analysis is required about which operation is more cost-efficient for each block recycling.

To identify the more cost-efficient operation between the migration and merge operations, we derive cost-per-benefit models for both operations. Let us assume that C_E is the block erase cost (or time) and C_{cp} is the page copy cost (or time) from a block to another. Many contemporary SLC Flash memory chips provide a *copyback* operation, which copies a page from a block to another through an internal buffer in the Flash memory chip. For those chips, a page copy could be done with a single *copyback* operation. Otherwise, as for MLC Flash memory chips, this would be done via two operations, that is, a page read and a subsequent page write.

To calculate the merge cost per available page, let us begin from when the merge operation has finished in Fig. 1(c) and a write request for sector 8 follows it. To service the write request, the FTL allocates and erases an empty log block with cost C_E. Then, all subsequent write requests are redirected to one of the N_p pages in the newly allocated log block. Later when all the pages are consumed, a merge operation is performed by erasing a new log block with cost C_E and copying all N_p pages from either the old data block or from the old log block with cost N_pC_{cp}. Then, the relevant map is updated and a period of block life cycle (interval between successive merge operations) ends. If a write request for sector 8~11 follows, then a new period of block life cycle begins again and the FTL will allocate an empty log block to serve that request. In total, the FTL pays merge cost, C_{merge}, to acquire N_p available pages for each period. Summing up, we can define merge cost as $2C_E + N_pC_{cp}$ and its benefit as N_p (available pages in the new log block) for each period. Hence, the merge cost per available page, W_{merge}, is as follows;

$$W_{merge} = \frac{C_{merge}}{N_p} = \frac{2C_E + N_p \cdot C_{cp}}{N_p}$$

To calculate migration cost per available page, let us again begin from when the migration operation has completed in Fig. 3(c) and a write request for sector 8 follows it. To service the write request, the FTL redirects the write request to one of N_p-p unused pages in the log block. After N_p-p write requests are served with that log block, either a migration or a merge operation should be performed. Let us assume that a migration operation is preferred. Then it will erase a new log block with cost C_E and copy p valid pages from the old log block to the new log block with cost pC_{cp}. In total, FTL pays migration cost, C_{mig}, to acquire N_p-p available pages for each period, where the migration cost, C_{mig}, is $C_E + pC_{cp}$. Hence, the migration cost per available page, W_{mig}, is as follows.

$$W_{mig} = \frac{C_E + p \cdot C_{cp}}{N_p - p}$$

These cost models for migration and merge operations do not consider the map update cost because it is almost identical for both operations and makes up only a small portion in the total block recycling cost. For example, our FTL is highly optimized to gather in a single map page all information that is to be modified during block recycling and, thus, it writes a single map page after block recycling.

With the cost models, an FTL can select a cost-effective operation for each block recycling. Note that the migration cost for an available page, W_{mig}, varies according to p, the number of copied pages during migration. At a certain value of p, W_{mig} becomes equal to W_{merge} and we call it the *equilibrium p*. The existence of an *equilibrium p* suggests that the migration operation be preferred when p is smaller than *equilibrium p* and vice versa.

$$\frac{2C_E + N_p \cdot C_{cp}}{N_p} = \frac{C_E + p \cdot C_{cp}}{N_p - p}$$

$$equilibrium\ \ p = \frac{N_p}{2}$$

5. MACROSCOPIC OPTIMAL MIGRATION/MERGE SEQUENCE

We implemented the cost-based selection between merge and migration operations and measured the performance with the Postmark benchmark. However, the performance improvements were much lower than we expected as will be shown later in Section 6. Also, better performance was observed by applying a merge operation even when p was much smaller than the *equilibrium p*. These observations lead us to recognize the existence of a macroscopic optimal migration/merge sequence that minimizes block recycling costs for each migration/merge combination period.

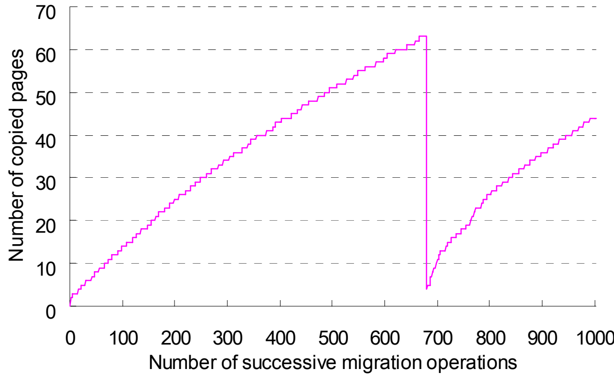

Figure 4. Number of copied pages during migration operation

Before we introduce the macroscopic optimization, we need to define P_n, a function for value p. Assume that a new period begins after a migration that copied p pages from an old log block to a new log block. In the next period, if only pages that already reside in the log block are written again, then the next migration will still copy p pages. On the other hand, if a write request for a new sector that does not exist in the log block arrives and is written to the log block, then the next migration will copy $p+1$ pages. In every case, the number of copied pages never decreases until a merge operation is performed; in fact, also it often increase as new sectors enter the log block. To confirm this analysis, we observed with the Postmark benchmark the number of copied pages of a block containing FAT and root directory sectors. The results, depicted in Fig. 4, show that the number of copied pages during migration increases almost linearly for successive migrations. This suggests a non-decreasing function P_n with an increasing rate α , where n is the number of successive migrations since an originating merge operation.

$$P_n = \alpha \cdot n$$

Let us now assume that a macroscopic period begins right after a merge operation and it ends by performing another merge operation. Also, we assume that migration operations are applied n times until the ending merge operation. Then, the objective becomes finding the n value such that the sequence minimizes the total block recycling costs for the macroscopic period, as described below.

Let us now elaborate using Fig. 5. In Fig. 5, N_p pages are available in a log block at the beginning of the macroscopic period. If all N_p pages are consumed, the FTL performs a migration operation that copies P_1 pages. This first migration produces N_p-P_1 pages available in the new log block, and now the FTL can serve N_p-P_1 write requests with those pages. Again if all pages in the log block are consumed, the FTL performs the second migration operation and produces N_p-P_2 pages available in the new log block. Similarly, migration operations are performed a total of n times and finally, a merge operation follows to end the macroscopic period. After the merge operation, all N_p pages are available in the new log block and another overall macroscopic migration/merge sequence begins. Now we define the macroscopic optimization as finding an optimal migration/merge sequence minimizing total block recycling costs for the macroscopic period. Specifically, when function P_n is given, the macroscopic optimization is identifying how many times the migration operations should be applied before the concluding merge operation by which the total block recycling cost becomes minimal.

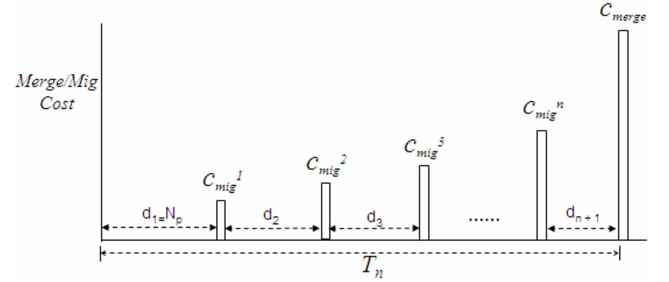

Figure 5. Migration/merge sequence

To figure out the optimal number of migration operations, we need to change some cost functions. With given P_n, $C_{mig}^{\,n}$, the n_{th} migration cost, can be defined as follows (see Fig. 3).

$$C_{mig}^{\;\;n} = P_n \cdot C_{cp} + C_E = \alpha \cdot n \cdot C_{cp} + C_E$$

After the *(n-1)*$_{th}$ migration, N_p-P_{n-1} pages are available in the new log block and d_n refers to the number of available pages after the *(n-1)*$_{th}$ migration.

$$d_n = N_P - P_{n-1} = N_P - \alpha(n-1), \;\; n >= 1$$

Now we can define *W(n)*, the migration/merge cost per an available page, for T_n, a macroscopic period of migration/merge sequence, as follows;

$$W(n) = \frac{\sum_{k=1}^{n} C_{mig}^{\;\;k} + C_{merge}}{\sum_{k=1}^{n+1} d_k} = \frac{\frac{\alpha \cdot C_{cp}}{2} n^2 + \left(\frac{\alpha \cdot C_{cp}}{2} + C_E\right) n + C_{merge}}{-\frac{1}{2}\alpha \cdot n^2 + \left(N_P - \frac{1}{2}\alpha\right) n + N_P}$$

As the macroscopic optimization of migration/merge sequence is finding n that minimizes *W(n)*, we differentiate *W(n)* to produce *W'(n)* and solve for n_0 that makes $W'(n_0) = 0$.[1]

$$W'(n) = \frac{\left(\frac{1}{2}C_E + \frac{1}{2}C_{cp} \cdot N_P\right)\alpha \cdot n^2 + \left(C_{cp} \cdot N_P + C_{merge}\right)\alpha \cdot n + \frac{\alpha \cdot C_{cp} \cdot N_P + \alpha \cdot C_{merge}}{2} + N_P \cdot C_E - N_P \cdot C_{merge}}{\left(-\frac{1}{2}\alpha \cdot n^2 + \left(N_P - \frac{1}{2}\alpha\right) n + N_P\right)^2}$$

[1] In order to differentiate, we assume that n and n_0 are real.

$$n_0 = \frac{-\left(C_{cp}\cdot N_P + C_{merge}\right)\alpha \pm \sqrt{\left(C_{cp}\cdot N_P + C_{merge}\right)^2\cdot\alpha^2 - \left(C_E + C_{cp}\cdot N_P\right)\cdot\alpha\cdot\left(C_{cp}\cdot N_P\cdot\alpha + \alpha\cdot C_{merge} + 2\cdot N_P\cdot C_E - 2\cdot N_P\cdot C_{merge}\right)}}{(C_E + C_{cp}\cdot N_P)\alpha}$$

This solution means that, when α is given, we can minimize the block recycling cost of a macroscopic period by performing migration operations n_0 times and then, subsequently perform a merge operation.

6. EXPERIMENTAL RESULTS OF MIGRATION/MERGE SEQUENCE

We implemented the migration operation by simply modifying the existing merge operation in our FTL that has been used commercially for MP3 players and camcorders. As a result, the FTL could select a cheaper operation between migration and merge for each block recycling. Also, the macroscopic optimization was implemented by estimating the increasing rate α of each log block with accumulated history of p. With the increasing rate α, the FTL calculates on-line the optimal number of migrations before a merge for each log block.

In our experiments, a benchmark or a workload-generating program runs on an MS-DOS FAT compatible file system and again the file system runs on our FTL. In turn, the FTL runs on a Flash memory emulator. The Flash memory emulator mimics the operations of Flash memory chips and also has an additional feature to calculate the elapsed time of Flash memory operations using time information provided in Flash memory chip datasheets. As we are interested in supporting high-capacity MLC Flash memory chips, typical Flash memory operation times specified in an MLC Flash memory datasheet were used in the experiments (Table 1). However, because the emulator ignores many chores encountered in real systems, the elapsed time of the emulator may differ from that of real systems.

To validate the elapsed time of the Flash memory emulator, we compared it with a measured elapsed time of a real embedded board that has a 200 Mhz S3C2410 (ARM920T core) processor and a 128 Mbyte NAND Flash memory chip.[2] As the elapsed time of the real board includes both Flash memory operation times and code execution times of test applications, the file system, and the FTL, we expected that the elapsed time of the real board would be larger than that of the emulator. Indeed, file read time of the board was greater than the emulator time as shown in Fig. 6. Surprisingly, however, file write time of the board exactly matched the emulator time. The reason was that actual page read time of the Flash memory chip was closely matched with that specified in the datasheet, but actual page write and block erase times were much smaller than those specified in the datasheet. As a result, in case of the file write test, the real board consumed less time for Flash memory operations than the emulator, but consumed additional time for code execution. These effects compensated for each other to generate almost the same file write time as shown in Fig. 6. From these observations, we can conclude that the emulator produces very realistic results for write operations, but underestimates for read operations.

[2] The Flash memory emulator used Flash memory operation times specified in the datasheet except for the bus transfer time, which was measured in a real board and used in the emulator.

Table 1. Typical Flash memory operation times [1]

Flash memory operation	Execution time (ms)
Page read (2112 bytes)	0.113
Page write (2112 bytes)	1.013
Page copy (2112 bytes)	1.128
Block erase	1.5

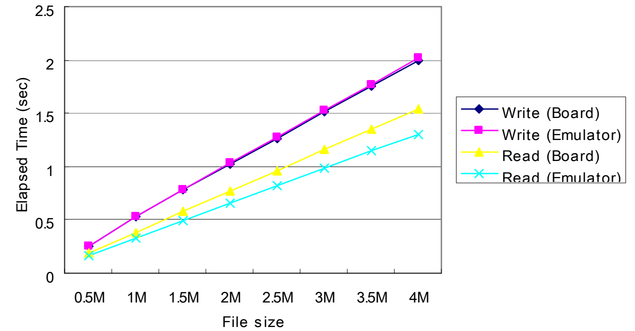

Figure 6. Elapsed time comparisons of emulator and real board

In the experiments, we used two test programs, the Postmark benchmark and embedded system workloads that had been used by Gal and Toledo [11]. With these test programs, we experimented with the following four schemes, that is, 1) merge only for block recycling, 2) cost-based selection between the migration or the merge operations, 3) cost-based selection with periodic merge, and 4) macroscopic optimization. In presenting the experimental results, these four schemes will be denoted as "merge-only", "merge+migration", "merge+migration (periodic merge)", and "merge+migration (optimization)", respectively. For the third scheme, the FTL counts the number of consecutive migration operations and if the count exceeds a predefined limit value, $N_p/2$ in our experiments, then it applies a merge operation even though the cost model recommends performing a migration operation. In other words, the "merge+migration (periodic merge)" is a blind version of "merge+migration (optimization)" because the former does not estimate the optimal number of migrations before merge, but simply guesses it.

Like the LFS, the performance of a page mapping FTL depends on the remaining free space. Also, the performance of the FAST scheme, which adopts sector mapping in log blocks, varies according to the number of log blocks. However, the performance of a block mapping FTL is not directly related to the number of log blocks except for when the number of log blocks is extremely small. In our experiments, only eight blocks were used as log blocks compared to the thousands of data blocks, and the performance was insensitive to the number of log blocks in this range.

6.1 Experiments with Postmark benchmark

The Postmark benchmark allows us to generate various workloads by setting parameters for behavior of transactions, number of transactions, numbers of files/directories to be created/deleted, file size, etc. We set those parameters to generate two different workloads that we will denote as "directory/small file workload" and "large file workload" in our experiments. In the former workload, the Postmark benchmark executes transactions that create/delete 50 directories and files of 512 bytes. In the latter workload, it executes transactions that create/delete 1 Mbytes files.

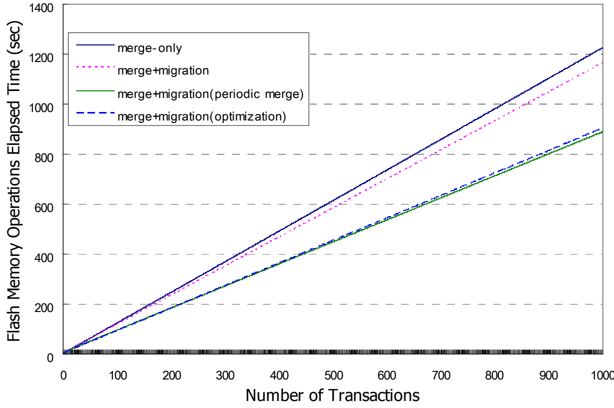

Figure 7. Elapsed times of Flash memory operations under "directory/small file workload" of Postmark benchmark

Fig. 7 shows that "merge+migration" improves performance about 5% over original "merge-only" scheme. Also, "merge+migration (periodic merge)" and "merge+migration (optimization)" improve performance up to 26~27% over "merge-only" scheme. In "merge+migration" scheme, the FTL executes the migration operation when p was smaller than 64 (=N_p /2), the *equilibrium p*; otherwise it executes the merge operation for block recycling. However, "merge+migration (optimization)" preferred merge to migration even when p was much smaller than 64. For example, in the Postmark benchmark, the increasing rate α of a log block for FAT and directory sectors was estimated to 0.1 and n_0 to 49, respectively, and the total block recycling costs became minimal by applying successively migration operations 49 times and then applying a merge operation and, at that time, p, the number of copied pages during migration, was about 5.

To investigate the benefits of the migration/merge sequence in detail, we analyzed the costs per available page of the "merge-only" scheme, W_{merge}, and the migration/merge sequence, $W(n)$ where n is the number of successive migration operations before merge (α =0.1). In Fig. 8, $W(n)$ plunges down as migration operations are applied for several times since the originating merge operation. It then becomes minimal when n is 49 and then increases slightly as n increases beyond the minimal point. At the minimal point, the migration/merge sequence cost is only 5% of the "merge-only" scheme for an available page.

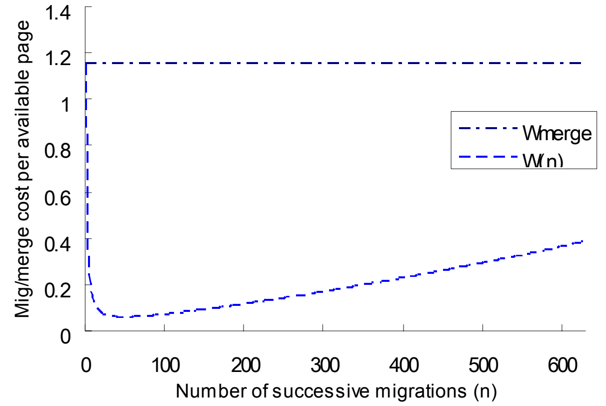

Figure 8. Cost per available page when α =0.1

W_{merge} represents cost-per-page of "merge-only" scheme and $W(n)$ represents cost-per-page of migration/merge sequence when migrations are performed successively n times before a merge operation is applied.

In our experiments, the "merge+migration (optimization)" calculates the optimal number of consecutive migrations on-line by estimating the increasing rate α while the "merge+migration (periodic merge)" fixes it with a predefined period value, $N_p/2$. In Fig. 7, surprisingly, the performance of "merge+migration (periodic merge)" is only a little better than "merge+migration (optimization)". We had chosen the period value, 64 (=$N_p/2$), arbitrarily, but further investigation showed that the chosen value was almost the off-line optimal for that workload. Fig. 9 shows the performance of "merge+migration (periodic merge)" for various period values from 20 to 120. In the figure, we observe that the off-line optimal performance of migration/merge sequence is around the period value 64 and it is better than that of the on-line "merge+migration (optimization)". However, the optimal period value may differ from workload to workload and the "merge+migration (periodic merge)" scheme would generally be blind to the workload characteristics. In Section 6.2, indeed, you can find that the blind "merge+migration (periodic merge)" performs worse than the workload adaptive "merge+migration (optimization)" for the embedded system workloads in which the predefined period value is not optimal any more. However, we should note that the "merge+migration (periodic merge)" scheme can be an alternative to "merge+migration (optimization)" if we already know the workload characteristics or if we want a trade-off between implementation overhead and performance.

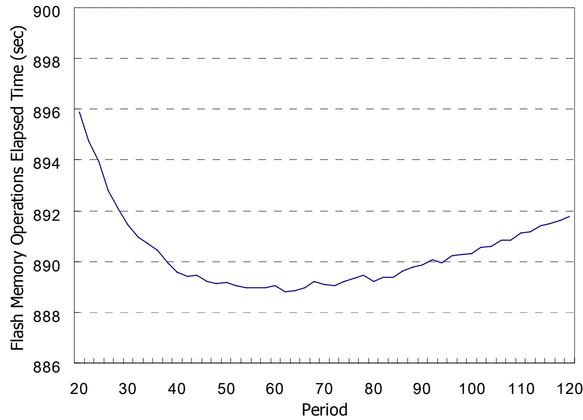

Figure 9. Elapsed times of "merge+migration (periodic merge)" scheme for various period values

Fig. 10 shows the performance of the four schemes under "large file workload" of the Postmark benchmark. Under the "large file workload", three schemes including migration performed better than the "merge-only" scheme, but the performance improvement rates were somewhat smaller than those under the "directory/small file workload". This is because most write activity of "large file workload" is sequential and that is not the target of the migration operation. However, some metadata such as FAT and directories are repeatedly modified while file data are sequentially written and the migration operation improves performance for those repeated updates of metadata as can be observed in Fig. 10.

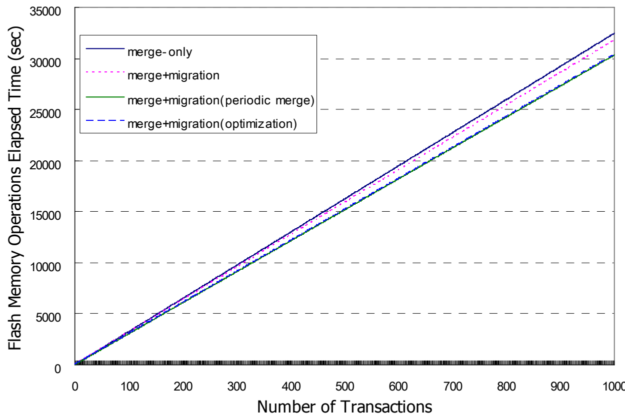

Figure 10. Elapsed times of Flash memory operations under "large file workload" of Postmark benchmark

6.2 Experiments with embedded system workloads

Embedded system workloads are a collection of Flash memory references of three different embedded devices such as a fax, a cellular phone, and an event recorder. Fig. 11 shows the performance of the four schemes under these workloads. In the figure, all workloads show a similar pattern in that performance increases more in the order of "merge-only", "merge+migration", "merge+migration (periodic merge)", and "merge+migration (optimization)" except for the one case with the event recorder where "merge-only" performs better than "merge+migration". Specifically, "merge+migration (optimization)" increases performance up to 5% over the "merge-only" scheme under the fax workload, 22% under the cellular phone workload, and 14% under the event recorder workload, respectively.

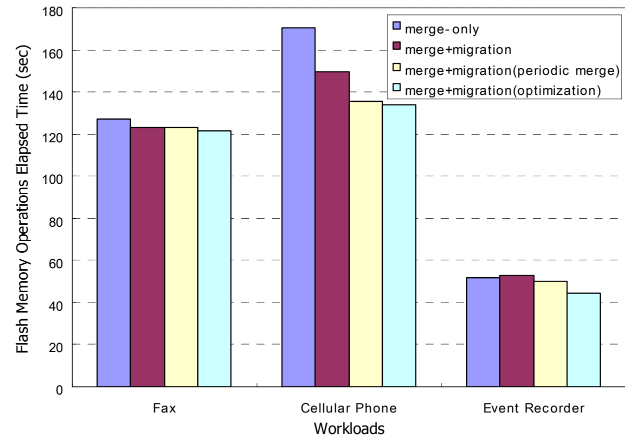

Figure 11. Elapsed times of Flash memory operations for the embedded system workloads

Under the event recorder workload, "merge+migration" performed slightly worse than "merge-only", and we investigated the reason behind this. Before the experiments, we formatted the file system and all the FAT sectors were written during the format. As a result, the log block for FAT sectors had already almost $N_p/2$ unique pages before the experiments. As the event recorder workload modifies only a small set of data repeatedly, only a few pages in the log block were repeatedly updated, but each migration copied almost $N_p/2$ pages all the time and produced $N_p/2$ available pages in the new log block. As only half of the pages are available in the new log block, block recycling incurred so frequently as to increase the burden of map updates and, as a matter of course, this is the worst case encountered by the "merge+migration" scheme.

If a merge operation was executed just once after the format, however, repetition of this worst case migration could be avoided. With the merge operation, the FTL would flush the already half-full log block and produce a whole-empty new log block. Thereafter then, each migration would copy only a small number of pages modified since the merge. In this case, the copy overhead and block recycling frequency would be much smaller than the previous case. This example explains why the "merge+migration (periodic merge)" and the "merge+migration (optimization)", which occasionally flush cold pages from log blocks with the merge operation, show better performance than the "merge+migration" scheme that does not. Also, under the embedded system workloads, the "merge+migration (optimization)" performs better than the "merge+migration (periodic merge)" because the former applies the merge operation at the best time determined via best-effort, while the latter does it

in a blind manner. These experimental results strongly suggest that a merge operation be applied among successive migrations to flush cold pages from log blocks, specifically when the workloads change.

7. CONCLUSION

As Flash memory has some inherent limitations, Flash memory based storage requires a software layer called the FTL that redirects modified data to a new location on Flash memory and recycles blocks with the so-called *merge* operation. In this paper, aside from the existing merge operation, we introduced another block recycling scheme that we call *migration*. Experiments with the Postmark benchmark and embedded system workloads show that a cost-based selection of migration or merge operation for each block recycling can reduce block recycling costs and therefore improve the performance of Flash memory based storage systems. Also, preliminary experiments suggested the existence of an optimal migration/merge sequence minimizing block recycling costs and, indeed, experiments with various workloads show that macroscopic optimization delivers much better performance than the simple cost-based selection. Finally, it should be noted that, as the macroscopic optimization suggests analytically, a periodic flushing of log blocks, that is, executing a merge operation among successive migration operations, is needed when combination of migration and merge is used for block recycling. In the future, we will investigate whether our scheme has a beneficial effect on other file systems such as the Linux Ext3 file system.

8. ACKNOWLEDGMENTS

This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software]] and supported in part by grant No. R01-2004-000-10188-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

We would also like to thank Prof. Sang Lyul Min of the Seoul National University for providing us with the Nikon Camera trace and other valuable advise.

9. REFERENCES

[1] 1G x 8Bit / 2G x 8Bit NAND Flash memory (K9L8G08U0M) Data Sheets, Samsung Electronics, Co., 2005.

[2] 512M x 8Bit / 256M x 16Bit NAND Flash Memory (K9K4Gxxx0M) Data Sheets, Samsung Electronics, Co., 2003.

[3] CF+ and CompactFlash Specification Revision 3.0, CompactFlash Association, 2004.

[4] Flash-Memory translation layer for NAND flash (NFTL), M-Systems.

[5] The MultiMediaCard System Summary, MMCA Technical Committee, 2005.

[6] Understanding the Flash Translation Layer (FTL) Specification, Intel Corporation, 1998.

[7] YAFFS (Yet Another Flash File System) Specification Version 0.3, http://www.aleph1.co.uk/yaffs/, 2002.

[8] Chiang, M.-L., Lee, P.C.H. and Chang, R.-C. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software: Practice and Experience*, *29* (3). 267-290.

[9] Chiang, M.-L., Lee, P.C.H. and Chang, R.C., Managing Flash Memory in Personal Communication Devices. in *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*, (1997), 177-182.

[10] Gal, E. and Toledo, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, *37* (2). 138-163.

[11] Gal, E. and Toledo, S., A Transactional Flash File System for Microcontrollers. in *USENIX Annual Technical Conference*, (2005), 89-104.

[12] Kang, J.-U., Jo, H., Kim, J.-S. and Lee, J., A Superblock-based Flash Translation Layer for NAND Flash Memory. in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, (Seoul, 2006), 161-170.

[13] Kawaguchi, A., Nishioka, S. and Motoda, H., A Flash-Memory Based File System. in *Proceedings of the Winter 1995 USENIX Technical Conference*, (1995), 155-164.

[14] Kim, J., Kim, J.M., Noh, S.H., Min, S.L. and Cho, Y. A Space-efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, *28* (2). 366-375.

[15] Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S. and Song, H.-J. A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation. *ACM Transactions on Embedded Computing Systems*, *6* (1).

[16] Rosenblum, M. and Ousterhout, J.K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, *10* (1). 26-52.

[17] Woodhouse, D. JFFS: The Journaling Flash File System *Ottawa Linux Symposium*, 2001.