

# Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design

Yuan-Hao Chang  
Graduate Institute of  
Networking and Multimedia  
Department of Computer  
Science and Information  
Engineering  
National Taiwan University  
Taipei 106, Taiwan (R.O.C.)  
d93944006@ntu.edu.tw

Jen-Wei Hsieh  
Department of Computer  
Science and Information  
Engineering  
National Chiayi University  
Chiayi 60004, Taiwan (R.O.C.)  
jenwei@mail.ncyu.edu.tw

Tei-Wei Kuo \*  
Graduate Institute of  
Networking and Multimedia  
Department of Computer  
Science and Information  
Engineering  
National Taiwan University  
Taipei 106, Taiwan (R.O.C.)  
ktw@csie.ntu.edu.tw

## ABSTRACT

This work is motivated by the strong demand of reliability enhancement over flash memory. Our objective is to improve the endurance of flash memory with limited overhead and without many modifications to popular implementation designs, such as Flash Translation Layer protocol (FTL) and NAND Flash Translation Layer protocol (NFTL). A static wear leveling mechanism is proposed with limited memory-space requirements and an efficient implementation. The properties of the mechanism are then explored with various implementation considerations. Through a series of experiments based on a realistic trace, we show that the endurance of FTL and NFTL could be significantly improved with limited system overheads.

## Categories and Subject Descriptors

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

## General Terms

Design, Experimentation, Management, Measurement, Performance, Reliability

## Keywords

flash memory, wear leveling, endurance, reliability

## 1. INTRODUCTION

While flash memory remains one of the most popular candidates in the storage-system designs of embedded systems, its application has now grown much beyond its original design goals. In recent years, it has become one part or layer in many system designs. Well-known examples are the flash-memory cache of hard

\*Supported by the National Science Council of Taiwan, R.O.C., under Grant NSC95R0062-AE00-07

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.

disks proposed by Intel [1, 9] and a fast booting service of Windows Vista by Microsoft [6, 9]. Such a development imposes harsh challenges on the reliability property of flash memory, especially on endurance. The reliability problem becomes even more challenging, especially when low-cost flash-memory designs gain their momentum in the market. For example, the endurance of a block of MLC<sub>×2</sub> flash memory is only 10,000 erase counts, compared to the 100,000 erase counts of its counterpart of SLC flash memory.<sup>1</sup> Although reliability is important, there is not much compromise possible for system performance and cost. These observations motivate the objective of this work. That is the exploration of the endurance issue of flash memory with limited overhead and without many modifications to popular implementation designs.

A NAND flash memory chip consists of many blocks, and each block is of a fixed number of pages. A block is the smallest unit for erase operations, while reads and writes are done in pages. Each page of small-block (/large-block) SLC flash memory can store 512B (/2KB) data, and there are 32 (/64) pages per block. The configuration of MLC<sub>×2</sub> flash memory is the same as large-block SLC flash memory, except that each block is composed of 128 pages [8]. Flash memory is usually managed by a block-device-emulating layer so that file systems could be built on its top (to be explained later with (Flash Translation Layer protocol (FTL) and (NAND Flash Translation Layer protocol (NFTL)). Such a layer implementation can be carried out by either software on a host system (as a raw medium) or hardware/firmware of the device for the flash. In the past decade, there are a number of excellent research and implementation designs to satisfy the performance needs of flash-memory storage systems, e.g., [2, 3, 4, 5, 11, 12, 17, 18]. Some explored different system architectures and layer designs, e.g., [11, 17], and some exploited large-scaled storage systems and data compression, e.g., [12, 18].

With the characteristics of flash memory in out-place updates, data that are to be updated must be written to another page of flash memory, where the original page of the data is marked as invalid. The out-place-update characteristic results in the wear-leveling issue over flash memory because any recycling of invalid pages will introduce block erasing. The objective of *wear leveling* is to have an even erase-count distribution of flash-memory blocks so that

<sup>1</sup>In this paper, we consider NAND flash memory, that is the most widely adopted flash memory in storage-system designs. There are two popular NAND flash memory designs: SLC (Single Level Cell) flash memory and MLC (Multiple Level Cell) flash memory. Each cell of SLC flash memory contains one-bit information, while each cell of MLC<sub>×n</sub> flash memory contains n-bit information.

the endurance of flash memory could be improved. In that direction, various excellent approaches based on dynamic wear leveling have been proposed, e.g., [7, 10, 11], where dynamic wear leveling achieves wear leveling by trying to recycle blocks with small erase counts. In such approaches, an efficient way to identify hot data (frequently updated data) becomes important, and excellent designs were proposed, e.g., [11, 13, 14, 15]. Although dynamic wear leveling does have great improvement on wear leveling, the endurance improvement is stringently constrained by its nature: That is, blocks of cold data are likely to stay intact, regardless of how updates of non-cold data wear out other blocks. In other words, updates and recycling of blocks/pages will only happen to blocks that are free or occupied by non-cold data, where cold data are infrequently updated data. *Static wear leveling* is orthogonal to dynamic wear leveling. Its objective is to prevent any cold data from staying at any block for a long period of time so that wear leveling could be evenly applied to all blocks.

Even though static wear leveling is very important to the endurance of flash memory, very little work is reported in the literature (except similar features mentioned in some products [7, 16]). In this paper, we propose a static wear leveling mechanism to improve the endurance of flash memory with limited memory-space requirements and an efficient implementation. In particular, an adjustable house-keeping data structure is presented, and a cyclic-queue-based scanning procedure is proposed for static wear leveling. Our objective is to improve the endurance of flash memory with limited overhead and without many modifications to popular implementation designs, such as FTL and NFTL. The behavior of the proposed mechanism is analyzed, with respect to FTL and NFTL, on main-memory requirements, extra block erases, and extra live-data copyings. A series of experiments is then conducted based on a realistic trace. We show that the endurance of FTL and NFTL could be significantly improved with limited system overheads. In particular, the endurance of FTL/(NFTL)-based storage systems could be improved by 51.2%/(87.5%) in terms of the first failure time of any blocks, and the distribution of erase counts over blocks was much improved.

The rest of this paper is organized as follows: Section 2 presents system architectures and summarizes popular existing implementations on Flash Translation Layer. In Section 3, an efficient static wear leveling mechanism is proposed. Section 4 provides analysis of the proposed mechanism, with respect to FTL and NFTL. Section 5 summarizes the experimental results on the endurance enhancement and extra overheads. Section 6 is the conclusion.

## 2. SYSTEM ARCHITECTURE

### 2.1 A Typical System Architecture of Popular File Systems

Consider a typical system architecture of flash-memory-based file systems, as shown in Figure 1. A Memory Technology Device (MTD) driver is to provide primitive functions, such as read, write, and erase over flash memory. A Flash Translation Layer driver is needed in the system for address translation and garbage collection, and FTL and NFTL are its popular implementations. A typical Flash Translation Layer driver consists of an *Allocator* and a *Cleaner*. The *Allocator* handles any translation of Logical Block Addresses (LBA) and their Physical Block Addresses (PBA). Different approaches have different address translation mechanisms, e.g., FTL and NFTL [3, 4, 5]. The *Cleaner* is to do garbage collection to reclaim pages of invalid data. Since garbage collection is done in the unit of a block, valid data in any pages of a block must be copied to other free pages when the block is to be erased. One

important implementation issue for flash-memory management is wear leveling, which is to evenly distribute the number of erasing for each block (because of the limitation on the number of erasing for blocks) so that the *SW Leveler* is proposed in this paper (Please see Section 3).

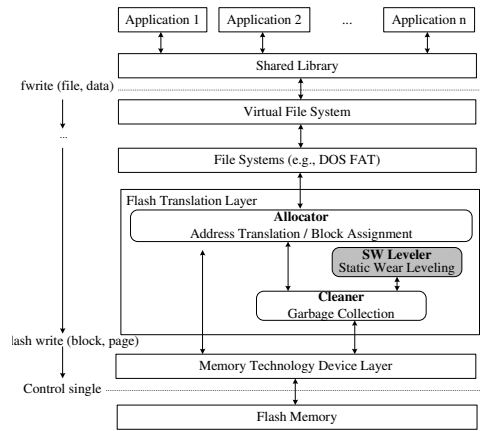


Figure 1: A Typical System Architecture of Popular File Systems

### 2.2 Existing Implementations on Flash Translation Layer

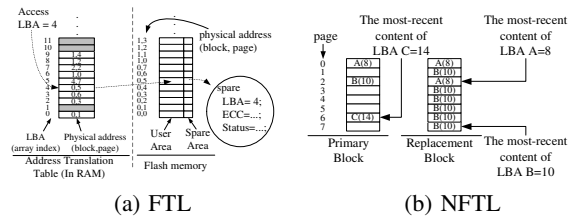


Figure 2: Flash Translation Mechanisms

FTL [2, 3, 5] adopts a page-level address translation mechanism for fine-grained address translation. The translation table in Figure 2(a) is one kind of fine-grained address translation. The LBA “4” is mapped to the physical block address (PBA) “(0,5)” by the translation table. LBAs are addresses of pages mentioned by the operating system, and each PBA has two parts, i.e., the residing block number and the page number in the block! When any data of a given LBA is updated, FTL must find a free page to store the data. When there is no sufficient number of free pages, garbage collection is triggered to reclaim the space of invalid pages by erasing their residing blocks. However, before a block is erased, data of any valid pages in the block must be copied to other free pages and have their corresponding translation table entries updated.

NFTL adopts a block-level address translation mechanism for coarse-grained address translation [4]. An LBA under NFTL is divided into a virtual block address and a block offset, where the virtual block address (VBA) is the quotient (i.e., that of the LBA divided by the number of pages in a block), and the block offset is the remainder of the division. A VBA can be translated to a (primary) physical block address by the block-level address translation. When a write request is issued, the content of the write request is written to the page with the corresponding block offset in the primary block. Since the following write requests can not overwrite

the same pages in the primary block, a replacement block is needed to handle subsequent write requests, and the contents of the (over-written) write requests are sequentially written to the replacement block. As shown in Figure 2(b), suppose that write requests to three LBAs  $A = 8$ ,  $B = 10$ , and  $C = 14$  are issued for 3 times, 7 times, and 1 time to a primary block and a replacement block, respectively, and the most-recent contents of  $A$ ,  $B$ , and  $F$  are also shown in the figure. When a replacement block is full, valid pages in the block and its associated primary block are merged into a new primary block (and a replacement block if needed), and the previous two blocks are erased. If there is not a sufficient number of free blocks, then garbage collection is triggered to merge valid pages of primary blocks and their corresponding replacement blocks so as to generate free blocks.

### 3. AN EFFICIENT STATIC WEAR LEVELING MECHANISM

#### 3.1 Overview

The motivation of static wear leveling is to prevent any cold data from staying at any block for a long period of time. It is to minimize the maximum erase-count difference of any two blocks so that the lifetime of flash memory is extended. In this paper, we shall consider a modular design for static wear leveling so that it can be integrated into many existing implementations with limited effort. As shown in Figure 1, we propose a static wear-leveling process for a Flash Translation Layer driver, referred to as the *SW Leveler*, to trigger the Cleaner to do garbage collection over selected blocks so that static wear leveling is achieved:

Let the SW Leveler be associated with a *Block Erasing Table (BET)* to remember which block has been erased in a selected period of time (Section 3.2). The SW Leveler is activated by some system parameters for the needs of static wear leveling (Section 3.3). When the SW Leveler is running, it either resets the BET or picks up a block that has not been erased so far (based on the BET information) and triggers the Cleaner to do garbage collection on the block. The selection procedure of a block must be done in an efficient way within a bounded amount of time. Note that the BET must be updated whenever a block is erased. It could be done by a triggering action to the SW Leveler. The design of the BET must be scalable because of the rapid increasing of flash-memory capacity and the limited RAM space on a controller. Whenever a block is recycled by garbage collection, any modification to the address translation is done as the original design of a Flash Translation Layer driver. The implementation of the SW Leveler could be a thread or a procedure triggered by a timer or the Allocator/Cleaner based on some preset conditions.

#### 3.2 Block Erasing Table

The purpose of the Block Erasing Table (BET) is to remember which block has been erased in a pre-determined time frame, referred to as the *resetting interval*, so as to locate blocks of cold data. A BET is a bit array, in which each bit corresponds to a set of  $2^k$  contiguous blocks where  $k$  is an integer that is larger or equal to 0. Whenever a block is erased by the Cleaner, the SW Leveler is triggered to set the corresponding bit as 1. Initially, the BET is reset to have 0 for every bit. As shown in Figure 3, there are one-to-one and one-to-many modes in the information maintenance, and one flag is used to track whether any one of the corresponding  $2^k$  blocks is erased. When  $k = 0$ , one flag is for one block (i.e., in the one-to-one mode). The larger the value of  $k$ , the higher the chance in the overlooking of blocks of cold data. However, a large value

for  $k$  could help in the reducing of the required RAM space of a controller for the BET.

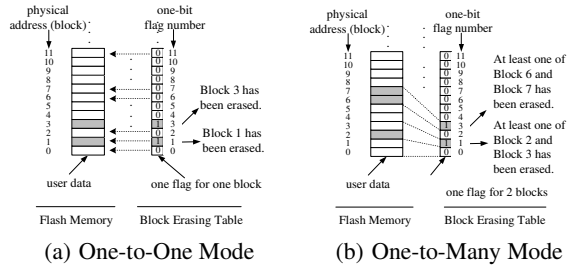


Figure 3: The Mapping Mechanism between Flags and Blocks

The worst case for a large  $k$  value occurs when hot and cold data co-exist in a block set. Fortunately, such a case is likely resolved sooner or later because pages of hot data will be eventually invalidated. As a result, cold data could be moved to other blocks by the SW Leveler (Please see Section 3.3). The technical problem relies on the tradeoff between the time to resolve such a case (bias in favor of a small  $k$ ) and the available RAM space for the BET (bias in favor of a large  $k$ ). Another technical issue is on the efficient rebuilding of the BET when a flash-memory storage system is attached. One simple but effective solution is to save the BET in the flash-memory storage system when the system shuts down and then to reload it from the system when needed. If the system is not properly shut down, we propose to load any existing correct version of the BET when the system is attached. Such a solution is reasonable as long as we do not skip too many times in the shutdown of the flash-memory storage system (or lose too much erase count information). Note that the crash resistance of the BET information in the storage system could be provided by the popular dual buffer concept. We shall also avoid the scanning of the spare areas of pages in the collection of any related information because of the potentially huge capacity of a flash-memory storage system.

#### 3.3 SW Leveler

The SW Leveler consists of the BET and two procedures in executing static wear leveling: SWL-Procedure and SWL-BETUpdate (Please see Algorithms 1 and 2). SWL-BETUpdate is invoked by the Cleaner to update the BET whenever any block is erased by the Cleaner in garbage collection. SWL-Procedure is invoked whenever static wear leveling is needed. Such a need is tracked by two variables, i.e.,  $f_{cnt}$  and  $e_{cnt}$ , where  $f_{cnt}$  and  $e_{cnt}$  denote the number of 1's in the BET and the total number of block erases done since the BET is reset, respectively. When the ratio of  $e_{cnt}$  and  $f_{cnt}$ , referred to as the *unevenness level*, is equal to or over a given threshold  $T$ , SWL-Procedure is invoked to trigger the Cleaner to do garbage collection over selected blocks such that cold data are moved. Note that a high unevenness level reflects the fact that a lot of erases are done on a small portion of the flash memory.

Algorithm 1 shows the algorithm of SWL-Procedure: SWL-Procedure simply returns if the BET is just reset (Step 1). While the unevenness level, i.e.,  $e_{cnt}/f_{cnt}$ , is over a given threshold  $T$ , the Cleaner is invoked in each iteration to do garbage collection over a selected set of blocks (Steps 2-12). In each iteration, it is checked up if all of the flags in the BET are set as 1 (Step 3). If so, the BET is reset, and the corresponding variables (i.e.,  $e_{cnt}$ ,  $f_{cnt}$ , and  $f_{index}$ ) are reset where  $f_{index}$  is the index in the selection of a block set for static wear leveling and is reset to a randomly selected block set (Steps 4-7). After the BET is reset, SWL-Procedure simply returns to start the next resetting interval (Step 8). Otherwise, the selection

---

**Algorithm 1: SWL-Procedure**

---

**Input:**  $e_{cnt}, f_{cnt}, k, f_{index}, BET$ , and  $T$   
**Output:**  $null$

```
1 if  $f_{cnt} = 0$  then return ;
2 while  $e_{cnt}/f_{cnt} \geq T$  do
  /*  $size(BET)$  is the number of flags in the BET. */
  3 if  $f_{cnt} \geq size(BET)$  then
    4    $e_{cnt} \leftarrow 0$ ;
    5    $f_{cnt} \leftarrow 0$ ;
    6    $f_{index} \leftarrow RANDOM(0, size(BET) - 1)$ ;
    7   reset all flags in the  $BET$ ;
    8   return;
  9 while  $BET[f_{index}] = 1$  do
    10    $f_{index} \leftarrow (f_{index} + 1) \bmod size(BET)$ 
  11 EraseBlockSet( $f_{index}, k$ ); /* Request the Cleaner to do
    garbage collection over the selected block set. */
  12  $f_{index} \leftarrow (f_{index} + 1) \bmod size(BET)$ ;
```

---

index, i.e.,  $f_{index}$ , moves to the next block set with a zero-valued flag (Steps 9-10). Note that the sequential scanning of blocks in the selection of block sets for static wear leveling is very effective in the implementation. We surmise that the design is close to that in a random selection policy in reality because cold data could virtually exist in any block in the physical address space of the flash memory. SWL-Procedure then invokes the Cleaner to do garbage collection over a selected block set (Step 11) and move to the next block set (Step 12). We must point out that  $f_{cnt}$  and the BET will be updated by SWL-BETUpdate accordingly because SWL-BETUpdate will be invoked by the Cleaner in garbage collection. The loop in static wear leveling will end until the unevenness level drops to a satisfiable value.

---

**Algorithm 2: SWL-BETUpdate**

---

**Input:**  $e_{cnt}, f_{cnt}, k, b_{index}$ , and  $BET$   
**Output:**  $e_{cnt}, f_{cnt}$  and  $BET$  are updated based on the erased block address  $b_{index}$  and  $k$  in the BET mapping.

```
1  $e_{cnt} \leftarrow e_{cnt} + 1$ ; /* Increase the total erase count. */
  /* Update the BET if needed. */
2 if  $BET[\lfloor b_{index}/2^k \rfloor] = 0$  then
  3    $BET[\lfloor b_{index}/2^k \rfloor] \leftarrow 1$ ;
  4    $f_{cnt} \leftarrow f_{cnt} + 1$ ;
```

---

SWL-BETUpdate is as shown in Algorithm 2: Given the address  $b_{index}$  of the block erased by the Cleaner, SWL-BETUpdate first increases the number of blocks erased in the resetting interval (Step 1). If the corresponding BET entry is not 1, then the entry is set as 1, and the number of 1's in the BET is increased by one (Steps 2-4). The remaining technical question is on the maintenance of the values of  $e_{cnt}$ ,  $f_{cnt}$ , and  $f_{index}$ . In order to do a better job in static wear leveling, their values should be saved on the flash memory as system parameters and retrieved in the attachment of the flash memory. However, we should point out that these values could tolerate some errors with minor modifications to SWL-Procedure on either the condition in Step 3 or the linear traversal of the BET (Steps 9-10). In other words, when the system crashes before their values are saved on the flash memory, we can simply use those saved in the flash memory previously.

## 4. PROPERTIES

The purpose of this section is to provide overhead/ performance analysis of the static wear leveling mechanism, with respect to well-known implementations. The experimental results of the mechanism will be summarized in Section 5.

### 4.1 Main-memory Requirements

	128MB	256MB	512MB	1GB	2GB	4GB
$k = 0$	128B	256B	512B	1024B	2048B	4096B
$k = 1$	64B	128B	256B	512B	1024B	2048B
$k = 2$	32B	64B	128B	256B	512B	1024B
$k = 3$	16B	32B	64B	128B	256B	512B

Table 1: The BET Size for SLC Flash Memory

Since one-bit flag is needed for each block set, the BET contributes the major main-memory space overheads on the controller so as to maintain the erase status of blocks. As shown in Table 1, the size of the BET varies, depending on the size of a flash-memory storage system and the value of  $k$ . For example, the BET size is **512B** for a 4GB SLC flash memory with  $k = 3$ . When MLC flash memory is adopted, the BET size will be much reduced. Note that although some implementations do have a larger page, such as 4KB pages in some NAND flash memory products designed by Samsung Co. Ltd., they usually have a huge capacity. The study on the BET size still provides good insight.

### 4.2 Extra Block Erases

Extra overhead in the recycling of blocks is, indeed, introduced by the proposed static wear leveling mechanism. A very minor part comes from the execution of SWL-BETUpdate whenever the Cleaner erases a block, i.e., the value updates of  $e_{cnt}$  and  $f_{cnt}$  and the flags of the BET (compared to the block erase time that could be about 1.5ms over a 1GB MLC<sub>x2</sub> flash memory [8]). As astute readers might point out, the Cleaner might be triggered more often than before because of static wear leveling. That might result in more block erases and live data copyings (to be discussed in the next sub-section).

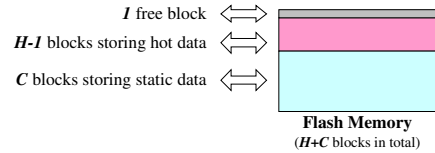


Figure 4: Flash Memory of only Cold Data and Hot Data

The overheads, due to extra block erases, could be better understood by some worst case study: The worst case on extra block erasing happens to the proposed mechanism when flash memory consists blocks of hot data, blocks of cold data, and exactly one free block in a resetting interval (as shown in Figure 4). Suppose that there are  $(H - 1)$  blocks of hot data and  $C$  blocks of cold data, where there are totally  $(H + C)$  blocks in the system. The situation becomes the worst when the  $C$  blocks are erased, only due to static wear leveling, and there are totally  $T \times (H + C)$  block erases in a resetting interval. Here  $T$  stands for the threshold value in the triggering of static wear leveling. This worst case happens when hot data are updated with the same frequency and only to the free block and the blocks of hot data, where  $k = 0$ . In each resetting interval, when the updates of hot data result in  $(T \times H)$  block erases, SWL-Procedure is activated to recycle one block of cold data for

the first time. Hereafter, SWL-Procedure is activated to recycle one block of cold data on every another  $(T - 1)$  block erases resulted by the updates of hot data. Finally, this procedure repeats for  $C$  times such that all BET flags are set, and a resetting interval ends. Among every  $T \times (H + C)$  block erases in a resetting interval,  $C$  block erases are done by SWL-Procedure, so the increased ratio of block erases (due to static wear leveling) is derived as follows:

$$\frac{C}{T \times (H + C) - C} \approx \frac{C}{T \times (H + C)}, \text{ when } T \times (H + C) \gg C.$$

The increased ratio looks even worse when  $C$  is a dominating part of  $(H + C)$  (where a study in [7] shows that the amount of non-hot data could be several times of that of hot data in many cases). Table 2 shows different increased ratios in extra block erasing for different configurations of  $H$ ,  $C$ , and  $T$ . As shown in the table, the increased overhead ratio in extra block erasing is sensitive to the setting of  $T$ . It shows that a good system setting should not have a very small  $T$  so that static wear leveling is triggered often.

H	C	H:C	T	Increased Ratio (%)
256	3840	1:15	100	0.946%
2048	2048	1:1	100	0.503%
256	3840	1:15	1000	0.094%
2048	2048	1:1	1000	0.050%

**Table 2: The Increased Ratio of Block Erases of a 1GB MLC<sub>×2</sub> Flash-Memory Storage System**

### 4.3 Extra Live-page Copyings

The extra overheads in live-page copyings, due to the static wear leveling mechanism, can be explored by the worst case study for the extra overheads in block erases (in the previous subsection). Let  $N$  be the number of pages in a block. Suppose that  $L$  is the average number of pages copied by the Cleaner in the erasing of a block of hot data. Thus, in the worst case, totally  $(C \times N)$  live-pages are copied on the erasing of  $C$  blocks of cold data, due to static wear leveling, in a resetting interval, and there are  $(T \times (H + C) - C) \times L$  live-page copyings because of regular activities of garbage collection in a resetting interval. The increased ratio in live-page copyings, due to static wear leveling, can be derived as follows:

$$\frac{C \times N}{(T \times (H + C) - C) \times L} \approx \frac{C \times N}{T \times L \times (H + C)}, \text{ when } T \times (H + C) \gg C.$$

H	C	H:C	T	L	$\frac{N}{T \times L}$	Increased Ratio (%)
256	3840	1:15	100	16	0.0800	7.572%
2048	2048	1:1	100	16	0.0800	4.002%
256	3840	1:15	100	32	0.0400	3.786%
2048	2048	1:1	100	32	0.0400	2.001%
256	3840	1:15	1000	16	0.0080	0.757%
2048	2048	1:1	1000	16	0.0080	0.400%
256	3840	1:15	1000	32	0.0040	0.379%
2048	2048	1:1	1000	32	0.0040	0.200%

**Table 3: The Increased Ratio in Live-page Copyings of a 1GB MLC<sub>×2</sub> Flash-Memory Storage System**

Table 3 shows different increased ratios of live-page copyings for different configurations of  $H$ ,  $C$ ,  $T$ , and  $L$ , when  $N = 128$ . The increased ratio of live-page copyings is sensitive to  $\frac{N}{T \times L}$  and  $T$ . As shown in Tables 2 and 3, the increased ratios of block erases and live-page copyings would be limited with a proper selection

of  $T$  and other parameters. They could be merely few percentages for FTL and NFTL implementations when static wear leveling is supported. In the next section, we shall further explore the performance of the proposed mechanism in static wear leveling, such as those on the endurance of flash memory and the unevenness level in the distribution of erase counts.

## 5. PERFORMANCE EVALUATION

### 5.1 Performance Metrics and Experiment Setup

The purpose of this section is to evaluate the capability of the proposed static wear leveling mechanism in FTL and NFTL, in terms of endurance (Section 5.2) and extra overhead (Section 5.3). The endurance metrics was based on the *first failure time* (i.e., the first time to wear out any block) and the *distribution of block erases*. The extra overhead was based on the percentage of extra block erases and extra live-page copyings, due to static wear leveling.

The Cleaners in FTL and NFTL w/o static wearing leveling all adopted the same greedy policy to have fair comparisons: That is, the erasing of a block with each valid page resulted in one unit of recycling cost, and that with each invalid page generated one unit of benefit. Block candidates for recycling were picked up by a cyclic scanning process over flash memory if their weighted sum of cost and benefit was above zero. (*Note that dynamic wear leveling was already adopted in the Cleaner of FTL and NFTL.*) The Cleaners in FTL and NFTL were triggered for garbage collection, when the percentage of free blocks was under 0.2% of the entire flash-memory capacity. Note that a primary block and its associated replacement block had to be recycled by NFTL when the replacement block was full.

1GB MLC<sub>×2</sub> flash memory (128 pages per block and 2KB per page) were under investigation. Note that FTL and NFTL could manage MLC flash memory devices with minor modifications. There were 2,097,152 LBAs for it. The experiment trace was collected over a mobile PC with a 20GB hard disk (by NTFS) for a month. The workload was mainly on daily activities, such as web surfing, email access, movie downloading and playing, game playing, and document editing. There were about 36.62% of LBAs being written in the collected trace, and the averaged number of write (/read) operations per second was 1.82 (/1.97). Accesses within the first 2,097,152 LBAs (i.e., sectors of disks) of the trace were used for the performance evaluation. In order to come out the first failure time of FTL and NFTL, a virtually unlimited experiment trace was also derived based on the collected trace by randomly picking up any 10-minute trace segment in the trace.

### 5.2 Endurance Improvement

Figure 5 shows that the proposed static wear leveling mechanism (referred to as SWL) resulted in significant improvement on the first failure time of FTL and NFTL, where the x-axis and y-axis of each sub-figure denotes the  $k$  value and the first failure time, respectively. For example, the improvement ratio on FTL reached 51.2% when  $T = 100$  and  $k = 0$ , and that on NFTL was 87.5%. In general, good improvement on NFTL was achieved with a small unevenness-level threshold  $T$  and a small  $k$  value because  $T$  and  $k$  affected the frequency and the BET resolution in static wear leveling, respectively.

It is interesting to see that good improvement on FTL was achieved with a small unevenness-level threshold  $T$  but a large  $k$  value. A large  $k$  value was favored in fine-grained address mapping with the consideration of the first failure time, and the improvement quickly saturated. It was because more data were moved at a time by SWL with a large  $k$  value so that better mixing of hot and non-hot data of

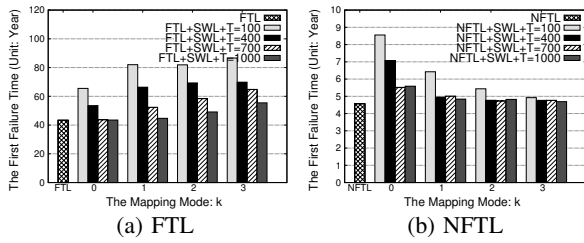


Figure 5: The First Failure Time

the trace let blocks be erased more evenly. Note that although the first failure time under FTL seemed long, it could be substantially shortened when flash memory is adopted in designs with a higher access frequency, e.g., disk cache. In addition, FTL is not practical in large-scale flash memory because it needs large main-memory space to maintain the address translation table.

	Avg.	Dev.	Max.
FTL	900	1118	2511
FTL + SWL + $k=0 + T=100$	930	245	2132
FTL + SWL + $k=0 + T=1000$	906	773	2361
FTL + SWL + $k=3 + T=100$	926	194	1708
FTL + SWL + $k=3 + T=1000$	900	884	2294
NFTL	9192	8112	20903
NFTL + SWL + $k=0 + T=100$	9234	609	11507
NFTL + SWL + $k=0 + T=1000$	9213	1738	15337
NFTL + SWL + $k=3 + T=100$	9288	5226	19520
NFTL + SWL + $k=3 + T=1000$	9314	7966	20617

Table 4: The Average, Standard Deviation, and Maximal Erase Counts of Blocks

Table 4 shows that SWL could greatly improve the deviation of block erases for FTL and NFTL, unless  $T$  and  $k$  both had large values. The experimental results were derived based on trace simulations of 10 years (Please see Section 5.1 on trace generation) even though some blocks were worn out. The maximal erase counts of FTL and NFTL were also significantly improved in a similar way. The results supported the improvement by SWL on the first failure time of FTL and NFTL.

### 5.3 Extra Overhead

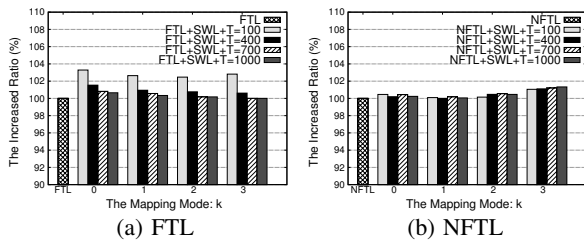


Figure 6: The Increased Ratio of Block Erases

Figure 6 shows the increased ratio of block erases on FTL and NFTL, due to static wear leveling. In general, extra overhead on block erases would be smaller for a large  $T$  value (because of the frequency in triggering SWL) and a large  $k$  value (because of the BET resolution so that the frequency in triggering SWL was lower). The increased ratio of block erases, due to SWL, on FTL (NFTL), was less than 3.5% (1%) in all cases.

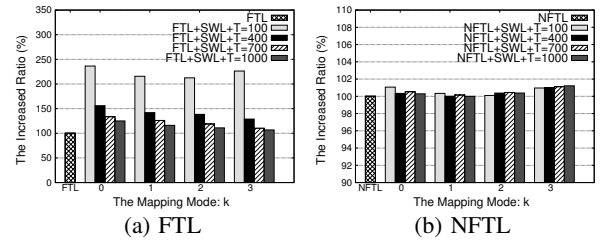


Figure 7: The Increased Ratio of Live-page Copyings

Figure 7 shows the increased ratios of live-page copyings, due to SWL, on FTL and NFTL. The increased ratio of live-page copyings, due to SWL, on NFTL was less than 1.5% in all cases. However, that on FTL was much larger because somehow hot data were often written in burst in the trace such that the average number of live-page copyings was very small under FTL. Because SWL moved data to achieve static wear leveling, a larger number of live-page copyings was introduced, compared to that under FTL without SWL.

## 6. CONCLUSION

This paper addresses a critical reliability issue in the deployment of flash memory in various system designs: The endurance of flash memory. Different from (dynamic) wear leveling explored in the previous work, we propose a static wear leveling mechanism to improve the endurance of flash memory with limited memory-space requirements and an efficient implementation. The results are critical, especially when flash memory are now deployed in system components with extremely high access frequencies, such as disk cache. We propose an adjustable house-keeping data structure and an efficient wear leveling implementation based on cyclic queue scanning to improve the endurance of flash memory with limited overhead and without many modifications to popular implementation designs, such as FTL and NFTL. A series of experiments shows that the endurance of FTL and NFTL could be significantly improved by more than 50% with limited system overheads.

For future research, we shall further explore the reliability enhancement problem of flash memory, especially when low-cost solutions, such as MLC, are adopted. We shall also exploit behaviors and issues of system components equipped with flash memory, such as those for external devices/adaptors.

## 7. REFERENCES

- [1] Flash Cache Memory Puts Robson in the Middle. Intel.
- [2] Flash File System. US Patent 540,448. In Intel Corporation.
- [3] FTL Logger Exchanging Data with FTL Systems. Technical report, Intel Corporation.
- [4] Flash-memory Translation Layer for NAND flash (NFTL). *M-Systems*, 1998.
- [5] Understanding the Flash Translation Layer (FTL) Specification. <http://developer.intel.com/>. Technical report, Intel Corporation, Dec. 1998.
- [6] Hybrid Hard Drives with Non-Volatile Flash and Longhorn. *Microsoft Corporation*, 2005.
- [7] Increasing Flash Solid State Disk Reliability. Technical report, SiliconSystems, Apr 2005.
- [8] NAND08Gx3C2A 8Gbit Multi-level NAND Flash Memory. *STMicroelectronics*, 2005.
- [9] Windows ReadyDrive and Hybrid Hard Disk Drives. <http://www.microsoft.com/whdc/device/storage/hybrid.mspx>. Technical report, Microsoft, May 2006.
- [10] A. Ban. Wear leveling of static areas in flash memory. US Patent 6,732,221. *M-systems*, May 2004.
- [11] L.-P. Chang and T.-W. Kuo. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.
- [12] L.-P. Chang and T.-W. Kuo. An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems. In *ACM Symposium on Applied Computing (SAC)*, pages 862–868, Mar 2004.
- [13] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29:3:267–290, May 1999.
- [14] J.-W. Hsieh, L.-P. Chang, and T.-W. Kuo. Efficient On-Line Identification of Hot Data for Flash-Memory Management. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 838–842, Mar 2005.
- [15] J. C. Sheng-Jie Syu. An Active Space Recycling Mechanism for Flash Storage Systems in Real-Time Application Environment. *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA'05)*, pages 53–59, 2005.
- [16] D. Shmidt. Technical note: Trueffs wear-leveling mechanism (tn-doc-017). Technical report, M-System, 2002.
- [17] C.-H. Wu and T.-W. Kuo. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *IEEE/ACM 2006 International Conference on Computer-Aided Design (ICCAD)*, November 2006.
- [18] K. S. Yim, H. Bahn, and K. Koh. A Flash Compression Layer for SmartMedia Card Systems. *IEEE Transactions on Consumer Electronics*, 50(1):192–197, February 2004.