

# 國立交通大學

資訊學院 資訊學程

碩士論文

應用於矽碟微控制器之平均磨損演算法的設計



Design and implementation of an efficient  
wear-leveling algorithm for solid-state-disk micro controllers

研究生：杜俊達

指導教授：張立平 博士

中華民國九十六年七月

應用於矽碟微控制器之平均磨損演算法的設計與實作

Design and implementation of an efficient  
wear-leveling algorithm for solid-state-disk micro controllers

研究生：杜俊達

Student：Chun-Da Du

指導教授：張立平 博士

Advisor：Dr. Li-Pin Chang

國立交通大學

資訊學院 資訊學程



Submitted to College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

July 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年七月

# 應用於矽碟微控制器之平均磨損演算法的設計與實作

學生：杜俊達

指導教授：張立平 博士

國立交通大學

資訊學院

資訊學程碩士班

## 摘要

隨著高容量的 NAND 快閃記憶體可以越來越負擔的起的時候，近來新興的應用之一是矽碟機用來取代手持裝置的硬碟。在實際上具有高度區域性工作量的矽碟機應用上，如果使用過去平均磨損演算法的適用性會被質疑。近年來研究人員提出了許多進階的平均抹損演算法。然而，就我們所知，目前沒有任何的進階平均抹損演算法可以實作在只具有少數資源的矽碟微控器上。本篇論文試著拉近演算法與實作的差距。我們的挑戰在於，如何使用有限的隨機存取記憶體空間，去追蹤記錄每一個快閃記憶體區塊抹損的次數。基於之前所提出的方法，Dual-Pool 演算法，我們只使用少於 200 位元的隨機存取記憶體，就可以有效的去平均抹損容量較大的 NAND 快閃記憶體。

# Design and implementation of an efficient wear-leveling algorithm for solid-state-disk micro controllers

student : Chun-Da Du

Advisors : Dr. Li-Pin Chang

Degree Program of Computer Science  
National Chiao Tung University

## ABSTRACT



As high-capacity NAND flash is becoming affordable, one among the recently emerging applications is to replace hard drives in mobile computers with solid-state disks (SSDs). The applicability of naive wear-leveling algorithms is largely concerned by realistic SSD workloads because the access patterns comprise strong spatial localities. Researchers have recently proposed advanced wear-leveling algorithms. However, to our best knowledge, there is little work regarding how the advanced algorithms can be realized in resource-restrictive SSD controllers. This work tries to close the gap between algorithms and implementation. The challenges pertain to how to use limited RAM space to keep track of the wearing of all the blocks. Based on a previously proposed, the dual-pool algorithm, we have shown that, by using no more than 200 bytes of RAM, wear leveling can be effectively conducted on a large NAND flash memory.

## 誌 謝

本篇論文的完成，首先要感謝指導教授張立平不誨的教導，因為我工作的關係，常常利用晚間以及週末的時間指導，我才能如期在兩年時間完成我的學業。另外要感謝我的家人，實驗室的同學，以及朋友們，你們的鼓勵是我最大的動力，謝謝大家。



# 目 錄

中文提要	.....	i
英文提要	.....	ii
誌謝	.....	iii
目錄	.....	iv
表目錄	.....	v
圖目錄	.....	vi
1、	Introduction.....	1
2、	Wear-Leveling Considerations .....	3
3、	Related Work.....	5
4、	Algorithm Design and Implementation .....	8
4.1	The Architecture of an SSD controller .....	8
4.1.1	Hardware Components .....	8
4.1.2	The Firmware .....	10
4.2	The Dual-Pool Algorithm .....	12
4.3	Algorithm Implementation .....	14
4.3.1	Data Structure and Operations .....	14
4.3.2	Local Wear Leveling .....	16
4.3.3	Adaptive Queue-Head Allocation Adjustment .....	19
4.3.4	Global Wear Leveling .....	20
4.3.5	Resource Utilization .....	21
5、	Experimental Results .....	22
5.1	Experimental Setup and Performance Metrics .....	22
5.2	Local Wear Leveling .....	23
5.3	Global Wear Leveling .....	25
5.4	Resource-Usage Optimization .....	27
5.4.1	Simulation validation .....	27
5.4.2	QH Table and Adaptive Queue-Head Allocation Adjustment .....	28
5.4.3	Effects of Large EH tables and QH tables .....	29
6、	Conclusion .....	30
References	.....	30

# 表 目 錄

Table1. A summary of wear-leveling algorithms considered in this paper .....	6
Table2. The five queue heads used by the dual-pool algorithm .....	14
Table3. Resource utilization of implementing the proposed wear-leveling algorithm .....	21
Table4. Standard deviation and overhead ratios of the third segment under different wear-leveling algorithm .....	24
Table5. Standard deviations and overhead ratios of all blocks of different segments under different settings .....	26
Table6. Response time of writes in seconds .....	27
Table7. Simulation validation using different types of workload .....	28
Table8. Static allocation and adaptive allocation of QH table entries among queue heads .....	28

# 圖 目 錄

Figure1. Translation logical address (FAT cluster numbers) to physical address (flash block numbers) .....	3
Figure2. A fragment of disk traces gathered from a mobile PC .....	4
Figure3. Examples of wear leveling .....	5
Figure4. The SSD evaluation board .....	8
Figure5. The block diagram of MC9S12UF32 .....	9
Figure6. Memory map of MC9S12UF32 .....	10
Figure7. The handling of block-device commands to an SSD .....	10
Figure8. The handling of a write to sector 35203 .....	11
Figure9. The concept of cold data migration. ....	13
Figure10. The concept of the dual-pool algorithm .....	15
Figure11. Relations between queue heads and operations DS/CPR/HPR .....	19
Figure12. The distribution of erasure cycles of the third segment under different wear-leveling algorithm .....	24



Figure13. Distribution of block-erasure cycles of all the  
blocks under global wear leveling ..... 26

Figure14. Effects of large EH tables and QH tables ..... 29



# 1 Introduction

One among the design challenges of mobile computers is that disk-based mass storage systems are power hungry and fragile to shock. Recently, vendors start replacing hard drives with NAND-flash-based solid-state disks<sup>1</sup> because flash memory is non-volatile, power-economic, shock-resistant, and free from positioning penalty. An SSD comprises a micro-controller and many NAND-flash chips [34]. It interacts with the host computer (e.g., a mobile PC) via standard interface such as ATA, SATA, or USB, as if it were a hard drive.

The physical characteristics of NAND flash impose unique challenges on the management of data. A NAND flash comprises an array of equal-sized blocks. Two consecutive writes to the same location on flash memory must be interleaved by erasure operations. An erasure operation wipes all data in a block. Because a block erasure would potentially involve valid data, necessary data-copy operations are performed before a block can be erased. Under the current technology, each individual block typically endures 100K erasure operations [35]. A worn-out block is not entitled for reliable data access.

In the past decade, NAND flash is widely used in multimedia appliances such as MP3 players and digital cameras. Such applications access usually exhibit very simple access patterns: Files are sequentially written onto flash memory and deleted in batches. The simple access patterns intrinsically give every NAND-flash blocks even chances to get erased. However, as mobile PCs start replacing hard drives with SSDs, NAND flash is exposed to workloads in which there are strong spatial localities [4, 22, 15]. With the presence of spatial localities, block erasure is usually directed to some particular blocks to reduce the overheads of data-copy operations. As a result, some blocks will be worn out quickly while the others remain fresh. Thus *wear leveling* is a notion to try to evenly distribute erasure operations over all the

---

<sup>1</sup>For example, Dell Latitude D420, Toshiba Dynabook SS RX1, Fujitsu LifeBook P1610, and Sony VAIO G1.

blocks. It is to postpone the first appearance of any bad block.

In the past decade, many excellent wear-leveling algorithms have been proposed from academia and industry [21, 30, 15, 9, 5, 17, 38]. Because wear leveling is concerned about to balance the wearing of blocks, one of the essential building block of wear-leveling algorithms is priority queue. On the other hand, it is necessary to persistently preserve wearing information of all the blocks. However, in [15] we have reported that prior work either require unaffordable computational resources [9, 17] or are ineffective in balancing the wearing of all the blocks. Additionally, to our best knowledge, there is no prior work regarding how wear-leveling algorithms can be realized in SSD controllers. The issue has its significance because not only SSD is the emerging application of NAND flash but also most SSD controllers are very restrictive in computational resources. For example, a typical SSD controller may be rated at 20 MHZ and has no more than 2 KB of embedded RAM [11, 28].

In this paper, we try to close the gap between advanced wear-leveling algorithms and the very resource-restrictive SSD controllers. A previously proven effective wear-leveling algorithm, the dual-pool algorithm [15], is considered in this paper. A bottom-up approach is taken for the realization of the dual-pool algorithm: First the basic building blocks, such as priority queues and data structures preserving block wearing information, are realized by using very limited resources. The technical questions mainly pertain to how to approximate the original concepts of the dual-pool algorithm while not introducing much performance distortion and management overheads.

The dual-pool algorithm is implemented in an SSD controller based on the building blocks. We have shown that the proposed implementation successfully conduct wear leveling over 8192 blocks of a 128 MB NAND flash by using no more than 200 bytes of RAM of an SSD controller. The resource-conservative version of the algorithm is as effective as it without resource constraints is. The algorithm itself consumes no noticeable SSD computation time. We believe that this paper is not only a piece of engineering work but also a justification of that wear leveling can be handed nicely by using very limited computational resources.

The rest of this paper is organized as follows: Section 2 introduces the characteristics of NAND flash and the considerations of wear leveling. Section 3 includes previously proposed wear-leveling algorithms. How the dual-pool algorithm is implemented in SSD controllers is

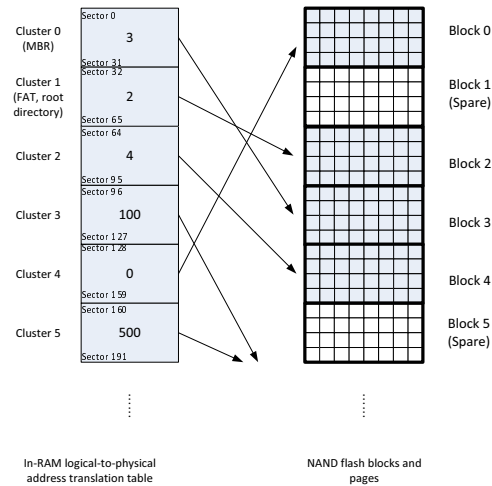


Figure 1: Translating logical addresses (FAT cluster numbers) to physical addresses (flash block numbers).

illustrated in Section 4, and Section 5 provides experimental results. This paper is concluded in Section 6.

## 2 Wear-Leveling Considerations

This section introduces the physical characteristics of flash memory and illustrate the needs of wear-leveling algorithms.

Physically, NAND flash is organized in terms of blocks, and each block is of a number of pages. A block and a page are the smallest units for erasure and read/write, respectively. In a typical NAND flash [35], a page is of 2048+64 bytes, and a block is of 64 pages. In particular, a page is divided into a user area (the part of 2048 bytes) and a spare area (the part of 64 bytes). The user area is the storage space of user data, while the spare area may contains out-of-band data such as ECC and some housekeeping information. Any two consecutive writes to the same page must be interleaved by a block erasure, which simultaneously wipes all pages of a block. Under the current technology, a block typically endures 100 K erasure operations until it becomes unreliable.

Because data-access patterns usually comprise spatial locality, to statically map a piece of data to a fixed flash-memory location would quickly wear out particular blocks. Instead, on each write the newly written data are directed to arbitrary free space. It is referred to as

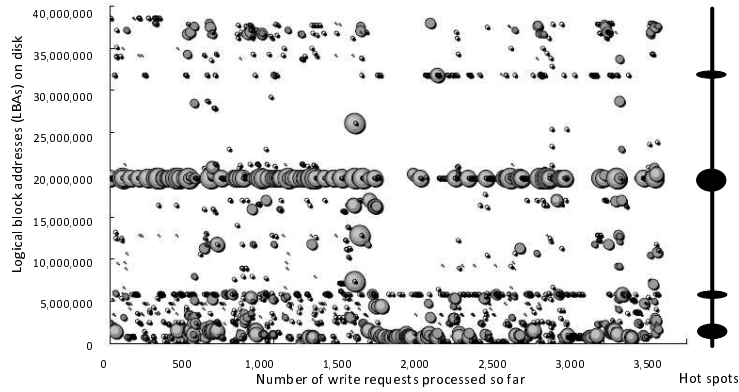


Figure 2: A fragment of disk traces gathered from a mobile PC. A large bubble refers to a large request.

“out-place update” is taken. Because where a piece of data now no longer stays in the same flash-memory locations, a mapping scheme is then needed to translate logical addresses into physical locations. As shown in Figure 1, the NAND flash emulate a block device, on which the FAT file system is mounted. Let the FAT cluster size be 16 KB, which is the same as the NAND-flash block size. Data are managed in terms of 16 KB, and the logical address and the physical address of a piece of data are the FAT cluster number and the NAND-flash block number, respectively. On each write to a cluster (e.g., cluster 0), first a spare block is found (e.g., block 1). The written data is then written to the spare block, and then the prior block of the cluster (e.g., block 3) is erased and becomes a spare block.

To solely use out-place cannot guarantee that every block is evenly worn. Realistic SSD workload expose NAND flash to strong spatial localities. Figure 2 shows a fragment of disk traces gathered from the daily use of a mobile PC. Each bubble in the figure refers to a write request, and the bubble size refers to the request size. As it shows, the majority of data are not even accessed. These data would long reside in blocks and thus the blocks does not participate the wearing of the entire flash memory. As indicated in the next section, data-migration policies can be adopted so as to give every block an even chance to get erased. Such activities are referred to as *wear leveling*. Figure 3 shows three scenarios of block wearing. Figure 3(a) indicates the case that no wear leveling is concerned, in which the distribution of erasure cycles is not even. Figure 3(b) shows that wear-leveling activities introduce too many overheads. Even though the erasure cycles are evenly distributed, overall the lifetime of

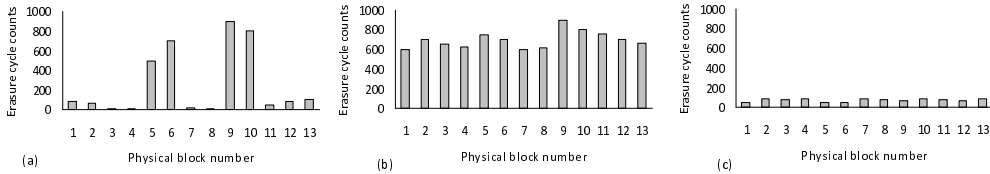


Figure 3: Examples of wear leveling. (a) No wear leveling or wear leveling has no effect. (b) Wear leveling introduces too much traffic. (c) A desirable result.

flash memory is consumed by data-migration introduced by wear leveling. Figure 3(c) shows a desirable case. All the erasure cycles are evenly distributed among blocks, and the cycle numbers are generally low.

Because wear leveling is concerned with the wearing of each block, ideally it is desirable to keep track of the wearing of all the blocks. Different from managing flash memory by using computational resources of the host computer (such as M-System’s NFTL [29], JFFS2 [5], and YAFFS [23]), to implement wear-leveling algorithms in SSD controllers, a severe challenge we faced is resource constraints. For example, a typical SSD controller is rated at 20 MHz. Many of prior work can not even be operational at this level of computing power [9, 17]. On the other hand, the controller has only 3.5 kilo bytes of RAM. Suppose that a 128 MB NAND flash is managed, in which there are 8192 16 KB blocks. As a large part of the RAM has been reserved to the translation table and block-device emulation, it is infeasible to keep the wearing information of all the blocks in RAM. However, to store the information on flash memory would significantly degrade the SSD performance, because to fetch data from NAND is significantly slower than to read from RAM. Besides, to frequently update the wearing information would unnecessary introduce extra write traffic to flash memory, and thus leads to more erasure operations.

### 3 Related Work

In the past decade, academia and industry contributed many wear-leveling algorithms. Table 1 summarizes some representative approaches. These algorithms can generally be divided into two categories: *Placement-based* algorithms and *reclaiming-based* algorithms. Let a block having a (relatively) large erasure cycle and a block having a small erasure cycle be referred

$\Delta$ : The maximum difference between any two block-erasure cycles

Algorithm Name	Block-Allocation Policy	Triggering Condition	Wear-leveling Policy	Reference
(HC) Hot-Cold Swapping	FIFO	Periodically	If $\Delta$ becomes larger than a predefined threshold, data stored in the oldest block and in the youngest block are swapped.	Chang et al. [10], Kim and Lee [7] M-System TrueFFS [17]
(2L) Two-Level	Youngest block first	Periodically	<ul style="list-style-type: none"> <li>● First level: To always allocate the youngest block for new writes.</li> <li>● Second level: If <math>\Delta</math> becomes larger than a predefined threshold, then move all live data away from the youngest block</li> </ul>	STMicroelectronics [25], Chang et al. [8]
(EP) Erase Pool	FIFO	On each write	Blocks of free space are organized as an erase pool, and newly written data are dispatched to free space allocated from blocks in the pool. How blocks are organized in the erase pool is undefined. FIFO is assumed in this paper.	SmartMedia[15], Sandisk [21]
(OP) Old-Block Protection	Youngest block first	On block erasure	<ul style="list-style-type: none"> <li>● If the difference between the erasure cycles of the <i>just-erased block</i> and the youngest block is larger than a predefined threshold, data in the youngest block are moved to the erased block.</li> <li>● If a just-erased block is involved in wear leveling, it won't be involved again until a predefined number of block erasures have been performed to other blocks.</li> </ul>	UBI [22]
(KL) Kim and Lee	Youngest block first	On garbage collection	Erase the block having the largest score for garbage collection according to Equation (2)	Kim and Lee [7]
(CAT) Cost-Age-Time	Youngest block first	On garbage collection	Erase the block having the largest score for garbage collection according to Equation (1)	Chiang et al. [20]
(TB) Turn-Based Selection	FIFO	Periodically	In $x$ out of $x+y$ times, garbage collection selects a block for erasure in favor of efficient garbage collection. In the rest $y$ times, a block is chosen for erasure according to some wear-leveling rules. In [7], a block of all live data is chosen, and in [8] a block is randomly selected for erasure <sup>2</sup> .	JFFS2 [13], YAFFS [14]
(DP) Dual-Pool	FIFO	Upon the completion of each write	<ul style="list-style-type: none"> <li>● Cold-data migration: use the migration of cold data to cease the wearing of overly worn blocks.</li> <li>● Hot-cold regulation: blocks are shielded from wear-leveling activities to see how the wearing of blocks introduced by cold-data migration develops.</li> </ul>	This paper

Table 1: A summary of wear-leveling algorithms considered in this paper.

to as an *old block* and a *young block*, respectively. The explanations of the algorithms are as follows:

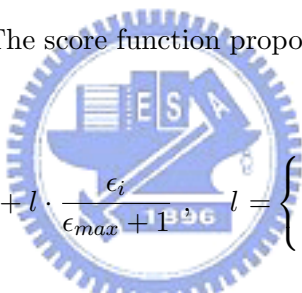
Placement-based algorithms accomplish wear leveling by means of to timely change data placement. Most of the algorithms are orthogonal to garbage collection. That is, they passively react to the wearing of blocks developed by garbage collection. Hot-cold swapping (algorithm HC) is proposed by M-System [30], Chang et al. [15], and Kim et al. [9]. It periodically swaps the data in the oldest block and in the youngest block. The idea is to “reverse” the wearing of such two kinds of blocks. Besides of swapping, blocks that have free space are organized in an FIFO queue. Newly written data are dispatched to the queue-head block, and newly erased blocks are appended to the queue. It is to fairly wear blocks. A different approach, algorithm 2L, tries to periodically “defrost” young blocks by moving all its data away. The approach is used by STMicroelectronics [38] and Chang et al. [13]. Bityutskiy et al. [21] (algorithm OP) proposed to check the youngest block and the block that has just been erased. If the latter block is old enough, then data in the youngest block are moved to the old block. The old block is then protected against being involved in wear leveling again until a number of block-erasure operations to any other blocks have been completed.

Reclaiming-based algorithms refer to garbage-collection policies that also take wear leveling into consideration. Turn-based selection (algorithm TB) is a commonly adopted technique

in the open-source community: In  $x$  out of  $x + y$  times, block erasure is handled in favor of efficient garbage collection. For the rest  $y$  times, block erasure is guided by some wear-leveling rules. In particular, JFFS2 [5] chooses a block of all live pages, and YAFFS [23] randomly picks a block for erasure <sup>2</sup>. The two approaches aim at providing young blocks some chances to get erased. Rather than to periodically switch between garbage collection and wear leveling, some algorithms choose to adaptively change their preferences by means of using a score function for blocks. The block of the largest score is always chosen for erasure. The score function proposed by Chiang et al. [17] (algorithm CAT) is

$$score(i) = \frac{\mu_i \cdot a_i(t)}{(1 - \mu_i) \cdot \epsilon_i} \quad (1)$$

, in which  $\mu_i$ ,  $a_i(t)$ ,  $\epsilon_i$  denote space utilization, an aging function, and the erasure cycle of block  $i$ , respectively. Function  $a_i(t)$  monotonically increases with  $t$  the time duration that how long block  $i$  has not been erased. The score function proposed by Kim and Lee [9] (algorithm KL) is



$$score(i) = (1 - l) \cdot \mu_i + l \cdot \frac{\epsilon_i}{\epsilon_{max} + 1} \quad l = \begin{cases} \frac{2}{1 + e^{\frac{k_\epsilon}{\Delta_\epsilon}}} & \text{if } \Delta_\epsilon \neq 0 \\ 0 & \text{, otherwise} \end{cases} \quad (2)$$

, in which  $\epsilon_{max}$  is the maximum erasure cycle of all blocks, and  $\Delta_\epsilon$  is the maximum difference between any two block-erasure cycles. Algorithm KL is configurable: the smaller  $k_\epsilon$  is, the more aggressive it would be for wear leveling. When the wearing of blocks becomes uneven, algorithm CAT and algorithm KL would prefer to erase blocks of small erasure cycles. They prefer efficient garbage collection when blocks are evenly worn.

Some algorithms do not explicitly conduct wear-leveling activities. In particular, approaches adopted by Sandisk [37] and SSFDC forum [36] simply rely on out-place update for “wear leveling”. However, this approach can not prevent cold data from being anchored in some particular blocks.

To our best knowledge, there is not yet related work on how wear-leveling algorithms can be implemented in resource-restrictive SSD controllers. In [15], it has been reported that

---

<sup>2</sup>Differences exist among revisions. The description is conceptual. YAFFS’s wear-leveling algorithm is proposed by its author but has not been officially integrated into the source tree.





Figure 4: The SSD evaluation board.

algorithm KL and algorithm CAT are too computation-intensive to be used in processors with weak computing power. All the other algorithms would require RAM space that can be afforded by SSD controllers.

## 4 Algorithm Design and Implementation

In this section, the architecture of the SSD controller is first explained. Based on its specifications and resource constraints, the design and implementation of the proposed wear-leveling algorithm is then introduced.

### 4.1 The Architecture of an SSD controller

#### 4.1.1 Hardware Components

The FreeScale M68KIT912UF32 development kit [28, 26] is chosen as our target platform. It is shown in Figure 4. On the evaluation board there is an SoC MC9S12UF32. As shown in Figure 5, the SoC comprises a 16-bit MCU core M68HC12, 3.5 KB of RAM, 32 KB of NVRAM (i.e., NOR flash), a USB 2.0 interface, flash-memory host controllers, and an integrated queue controller (i.e., IQUEUE) with 1.5 KB QRAM buffer. The SoC aims at products including USB thumb drives, card readers, and solid-state disks.

The MC9S12UF32 (referred to as the SSD controller in the rest of this paper) accepts various types of flash-memory cards, including Security Digital (SD), Multi Media Card (MMC),

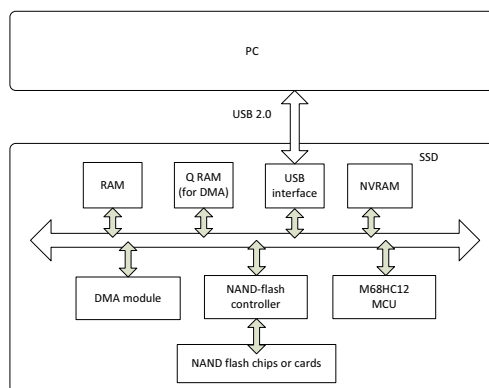


Figure 5: The block diagram of MC9S12UF32.

Smart Media (SM), and Memory Stick (MS). It can also access ATA-based storage devices such as Compact Flash (CF) and ATA hard drives. We choose to use SM cards because, other than SM and MS cards, all the flash-memory cards hide the physical geometry of NAND flash from outside (i.e., another controller sits inside the card). An SM card is actually bare NAND flash and its blocks and pages are accessible to the SSD controller. The SSD controller interacts with the host via USB 2.0. It supports USB endpoints of control, interrupt, isochronous, and bulk. The SSD interacts with the host computer as if it were a hard drive.

The flash-memory controllers and other components are controlled by the MCU by means of their I/O registers. The registers are mapped into the MCU memory space. The memory mapping is shown in Figure 6. The firmware stored in the NVRAM is mapped to two 16 KB regions of the MCU memory space. Between addresses 1200h and 2000h, 1084 bytes of RAM are reserved as runtime memory, and two 1 KB regions are set aside for flash-memory management.

To achieve high throughput, all the data transfers between flash memory and the USB interface are carried out by a DMA module, e.g., the IQUEUE. The IQUEUE has eight DMA channels. A 1.5 KB buffer between 2000h to 2600h is reserved for DMA operations, as Figure 6 shows. The design is to exploit potential parallelism. For example, while data are read from NAND flash to QRAM, it is possible to concurrently transfer data to the USB interface. All the high-speed data transfers need no intervention from the MCU.

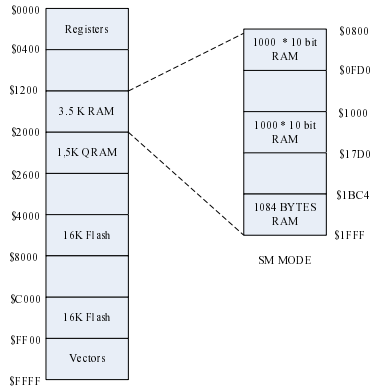


Figure 6: Memory map of MC9S12UF32.

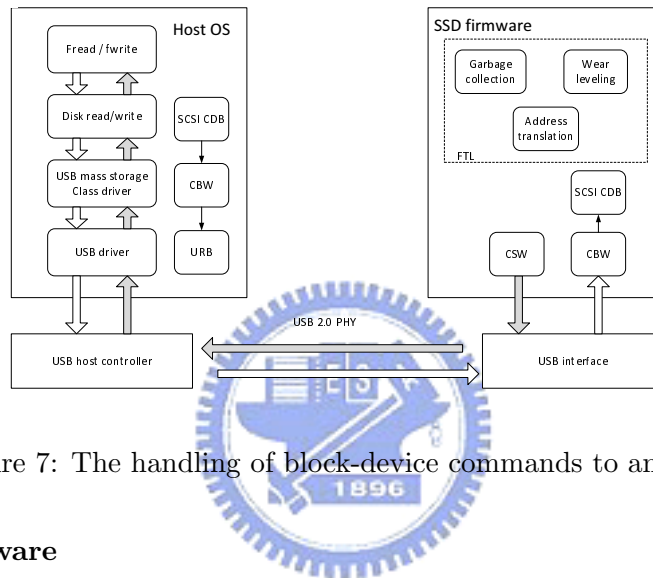


Figure 7: The handling of block-device commands to an SSD.

#### 4.1.2 The Firmware

The firmware has two major tasks: block-device emulation and block-device command processing. Flash-memory Translation Layer (i.e., FTL) is implemented for block-device emulation. It involves address translation, garbage collection, and wear leveling. The firmware accepts incoming block-device commands, translates disk geometry into flash-memory physical locations, handles garbage collection and wear leveling, and responds to the host upon the completion of data transfer.

Because the SSD is a USB device, it is compliant to the USB mass-storage class specification. As shown in Figure 7, to read or write a sector of the SSD, the host first composes a block-device command (e.g., SCSI read 28h or SCSI write 2Ah) as a SCSI command descriptor block (CDB). The CDB is then encapsulated by a USB command block wrapper (CBW). The CBW has USB-specific information such as the data-transfer direction (IN or OUT). The CBW is then encompassed by a USB Request Block (URB), which is sent to the USB

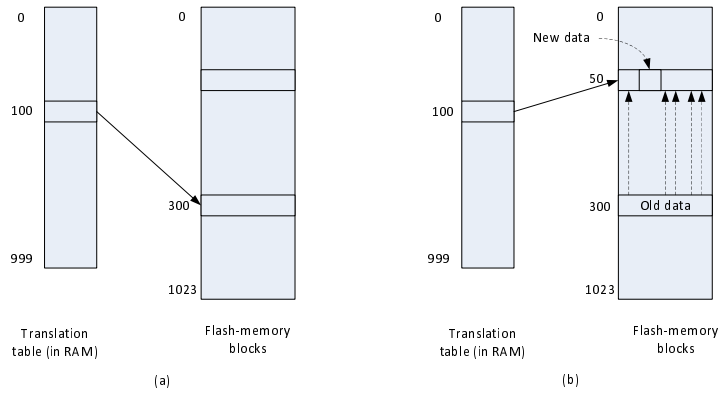


Figure 8: The handling of a write to sector 35203.

driver of the host. The CBW is then received by the SSD via the USB PHY, and the CDB is extracted from the CBW. The block-device command CDB is then taken over by the FTL for flash-memory operations. Upon the completion, the firmware composes a command status wrapper (CSW) for response. The CSW is sent to the host via the USB PHY, and the host USB driver extracts necessary information from the CSW.

The FTL needs to handle address translation, garbage collection, and wear leveling. Suppose that 128 MB NAND flash is used. Let a 128 MB NAND flash be considered. Let the page size and the block size be 512 bytes and 16K bytes, respectively. The flash memory is logically divided into 8 segments, each of which has 1024 16-KB blocks. The SSD exposes itself as a  $1000 \times 16K \times 8$  bytes block device. The firmware adopts a two-level mapping mechanism: At the first level, the eight  $1000 \times 16K$  bytes are statically mapped to the eight segments. At the second level, the  $1000 \times 16K$  bytes are mapped to 1024 blocks. Let the mapping unit size be 16K bytes, so in each segment 1000 logical units are mapped to 1024 physical units. There are  $1024 - 1000 = 24$  spare blocks for garbage collection and bad-block management. For the second-level mapping, each of the eight segment needs a translation table to translate a logical-unit address to the corresponding physical-unit address. A table is of 1000 entries and each entry refers to one of the 1024 physical units. To save RAM space, only two translation table are cached in RAM (address 0800h to 0fd0h and 1000h to 7d0h), as shown in Figure 6.

Because a mapping unit is 16 KB, which is no smaller than the block size, any write smaller than 16 KB rewrites an entire mapping unit because no partial update is allowed. Garbage-collection policy becomes trivial in this case. Let us consider an example shown in Figure

8. Suppose that a write to sector 35203 is received (the sector size is 512 bytes). First the corresponding segment is identified by

$$(35203 * 512) / (1000 * 16 * 1024) = 1$$

. So the data falls in segment 1. Let  $T_1[]$  be the translation table of segment 1. The data reside in block address

$$T_1[((35203 * 512) \% (1000 * 16 * 1024)) / (16 * 1024)] = 300$$

of segment 1. Inside block 300, the data is in page

$$(((35203 * 512) \% (1000 * 16 * 1024)) \% (16 * 1024)) / 512 = 3$$

. Let spare block 50 is used. Along with the new data, all the old data of block 100 except that in page 3 are copied to block 50, and then  $T_1[300]$  is assigned to 50. Block 100 is erased and becomes a spare block.

To conduct wear leveling over blocks in a segment, on write a block is allocated from the 24 spare blocks in a FIFO fashion. There is no wear leveling over blocks of different segments. In later sections we shall show that this approach is quite ineffective.



## 4.2 The Dual-Pool Algorithm

This section briefly introduce the concepts of the dual-pool algorithm. Before the explanations, some terminologies are in order: The erasure cycle of a block refers to how many times a block has been erased. Let a block having a relatively large erasure cycle and that having a relatively small erasure cycle be referred to as an old block and a young block, respectively. Cooling down a block and warming up a block means to start wearing the block and to cease its wearing, respectively. The dual-pool algorithm, as implied by its name, has a cold pool and a hot pool. They are logic aggregations of blocks. The cold pool can be considered as a place where blocks being cooled down gather. The hot pool is a place where blocks being warmed up gather. Initially blocks arbitrarily joins one of the two pool.

The basic ideas behind the dual-pool algorithm is *cold-data migration* and *hot-cold regulation*. Based on realistic SSD workloads, the majority of data are cold. Thus the cold data statically

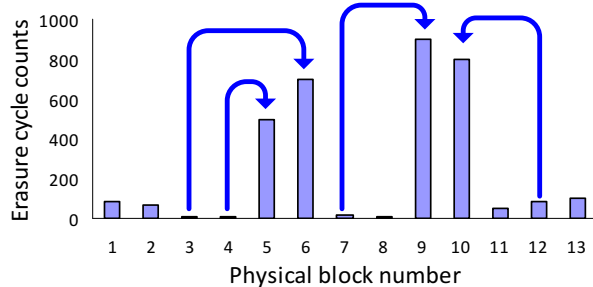


Figure 9: The concept of cold-data migration. Cold data are moved from young blocks to old blocks to cease the wearing of old blocks.

reside in a large number of blocks. These blocks are never preferred by garbage collection because to erase them involve heavy overheads. As shown by Figure 9, these blocks would have relatively low erasure cycles. On the other hand, a small number of blocks would frequently participate the circulation of hot data, as hot data are frequently updated in an out-place fashion. As the life cycle of hot data are very short, these blocks would quickly be filled up with invalid data and thus be good candidates for garbage collection. Different from prior work which tries to start wearing the young blocks, the dual-pool algorithm chooses to move cold data from young blocks into old blocks to cease the wearing of old blocks, as illustrated in Figure 9. It is cold-data migration. The rationale is that there are sufficiently many cold data to quench the wearing of a small number of blocks. By using prior approaches, because cold data is the majority, to move cold data around actually pointlessly circulate cold data among young blocks.

If a block has just been involved in cold-data migration, it should be shielded from being involved again. It is to develop the effects of cold-data migration, because the effects are not immediate. It is hot-cold regulation. To realize the idea, an old block to which cold data have just been moved in is thrown into the cold pool. A young block from which cold data just have been moved out is thrown into the hot pool. The operation is called *dirty swap* (or **DS** for short). Right after dirty swap, the two blocks are hidden in the two pool. As the effects of cold-data migration develop, the prior old block and the prior young block become young and old, respectively. They will eventually be eligible for next cold-data migration. The idea is shown in Figure 10.

The above algorithm is fragile if spatial localities move around. A block in the hot pool is

Symbol	Pool Association	Used by
$H^+(Q_{HP}^{EC})$	The hot pool	DS and HPR
$H^-(Q_{HP}^{EC})$	The hot pool	HPR
$H^+(Q_{HP}^{EEC})$	The hot pool	CPR
$H^-(Q_{CP}^{EC})$	The cold pool	DS
$H^+(Q_{CP}^{EEC})$	The cold pool	CPR

Table 2: The five queue heads used by the dual-pool algorithm.

intended to be warmed up. If the data in it become cold, the block is then stop being erased. Consequently the blocks deeply “sinks” in the hot pool and is never involved in DS. The same thing happens to the cold pool. To deal with this problem, two operations, cold-pool resize (CPR for short) and hot-pool resize (HPR for short) are introduced. They are to identify and to correct incorrect associations between pools and blocks. As Figure 10 show, a block storing cold data in the hot pool will be identified and moved to the cold pool by means of CPR. New block-wearing information of blocks, referred to as effective erasure cycles, is introduced. An effective erasure increases as a erasure cycle. However, they are reset to zeros as the corresponding block is involved in DS. The information is to help CPR to work correctly. The dual-pool algorithm is concerned with five priority queues. A priority queue prioritizes blocks in a pool in terms of a particular wearing information. Let  $Q_Y^X$  be a priority queue that prioritizes blocks in pool  $X$  in terms of wearing information  $Y$ . Let  $H^+(Q_Y^X)$  and  $H^-(Q_Y^X)$  be the maximum queue head (i.e., a block) and the minimal queue head of queue  $Q_Y^X$ . Let functions  $EC(B)$  and  $EEC(B)$  returns the erasure cycle and the effective erasure cycle of queue-head block  $B$ . A summary of the symbol definitions can be found in Table 2. Detailed description of the dual-pool algorithm can be found in [15].

### 4.3 Algorithm Implementation

#### 4.3.1 Data Structures and Operations

Consider a 128 MB flash of 16 KB blocks, in which there are 8192 blocks to manage. It is infeasible to even put one priority queue in the RAM of the SSD controller. New data structures are designed to overcome this restriction. The basic idea is to keep only frequently needed information in RAM, and to leave the rest on flash memory. The following are definitions of



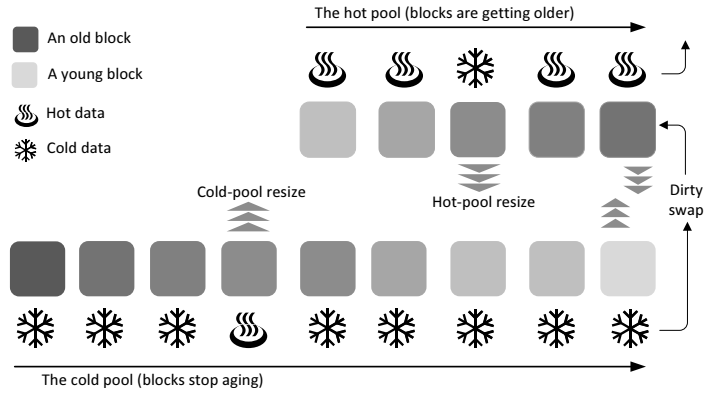


Figure 10: The concept of the dual-pool algorithm.

the data structures:

- On-flash data structures

**Block-Wear Table (BW table):** In the dual-pool algorithm, a block is associated with wearing information including its erasure cycle, its effective erasure cycle, and its pool association. The information all blocks in a segment is recorded in an erasure-cycle table (i.e., BW table). In the BW table, an entry for a block is of a 18-bit erasure cycle, a 13-bit effective-erasure cycle, and one bit pool association. The effective-erasure cycle is 13 bits because it is frequently reset (upon dirty swap) and need not to be large. There are 1024 blocks in a segment and each of which uses 4 bytes as its wearing information, so a block sufficiently accommodates the entire BW table of the segment. Note that the block can be arbitrary one in the segment. Note that the wearing information is not written to the spare area of any page because 1) to leave the spare areas intact for maximum compatibility, and 2) to avoid multiple writes to a spare area, which are prohibited in many NAND flash. The second case happens because a spare area will later be written to the error-correction codes.

- RAM-resident data structures:

**Erasure-History Table (EH table):** The BW table on flash memory needs to be revised on the completion of each block erasure. The overheads are apparently unaffordable because it doubles the traffic to flash memory. Instead, we reserve a small buffer in RAM (i.e., the erasure-history table, EH table) to log the blocks that are recently erased. Whenever the EH



table is full, the information in the EH table is merged with the BW table on flash, and a new BW table is written onto flash. Currently the EH table is of eight entries.

**Queue Head Table (QH table):** Because it is infeasible to keep an entire priority queue in RAM, an approximation is to keep only a number of queue heads of the queue. A table of 10 queue-head entries are reserved, and initially each priority queue share 2 entries of the table. Each entry is of 6 bytes that records a queue-head block number and the wearing information of the block. For example, the queue-head table has the first two minimal queue-head blocks of priority queue  $Q_{CP}^{EC}$ . Block  $H^-(Q_{CP}^{EC})$  refers to one of the two queue heads.

**Address Translation Table (AT table):** As shown in Figure 8, a piece of data may change its physical locations on update. A translation table resides in RAM to translate the logical address of a mapping unit to its physical addresses.

- Segment operations:

**Segment Check-in:** Because of RAM-space constraints, the controller's RAM is not sufficiently large to accommodate the RAM-resident data structures of all the segments. Alternatively, only that of two segments are kept in RAM. Whenever a SCSI read or write command referring to a segment and data structures of which are not cached in RAM, all the RAM-resident data structures are constructed in RAM. It is referred to as segment check-in. The following steps are taken to check in a segment: 1) The spare area of the first page of every block is scanned to construct the AT table, 2) the block storing the BW table is fetched and scanned to construct the QH table, and 3) the EH table is initialized as empty.

**Segment Check-out:** A segment may be replaced and all its data structures need to be committed on to flash memory. It is referred to as segment check-out. To check out a segment, the following steps are taken: 1) The on-flash BW table is merged with the EH table and a new BW table is written to a spare block and 2) the QH table is refreshed according to the new BW table, and 3) the EH table is initialized as empty. Because the frequently updated file-system structures like FATs reside in the first 1000\*16K bytes of the block device, the corresponding segment's data structures are always pinned in RAM.

**BW-table Merge:** As mentioned above, an old BW table is merged with the EH table as a new BW table. To do this, a spare block is first located. Pages of the block storing the BW

table are sequentially copied to the spare block. As a page is fetched by means of DMA and is temporarily buffered in the QRAM, necessary updates are made according to the EH table. The page is then written to the spare block. The merge operation repeats until all pages are copied.

### 4.3.2 Local Wear Leveling

Because a piece of data can not be mapped to different segments, basically wear leveling is confined to blocks of the same segment. Local wear leveling refers to wear-leveling activities conducted over blocks in the same block.

If the SSD is used at the first time, the first 512 blocks and the next 511 blocks are associated with the hot pool and the cold pool, respectively. The last block stores an empty BW table. As a segment is checked in, local wear leveling for the segment is activated, until it is checked out. On the completion of a block erasure, DS, HPR, and CPR are then in turn invoked to examine whether any actions must be taken. the number of the block is inserted to the EH table. The wearing information of the blocks in the QH table are revised accordingly. If the EH table is full, then a segment check-out is forced. The guideline to revise the dual-pool algorithm is to confine the wear-leveling activities to the queue-head blocks in the QH table. Interactions among the EH table, the QH table, and the on-flash BW table need some attention.

Let  $S$  be the collection of spare blocks of a segment. The followings are the revised dual-pool algorithm:

**Dirty Swap (DS):** On the completion of a block erasure, the following condition is checked:

$$EC(H^+(Q_{HP}^{EC})) - EC(H^-(Q_{CP}^{EC})) > TH.$$

If the condition is true, the following procedure is performed:

**Step 0a.** Allocate a spare block  $s_b$  from  $S$ .  $S \leftarrow S \setminus \{s_b\}$ .

**Step 1.** Copy data in block  $H^+(Q_{HP}^{EC})$  to spare block  $s_b$ .

**Step 2.** Erase block  $H^+(Q_{HP}^{EC})$ .

**Step 3.** Copy data from block  $H^-(Q_{CP}^{EC})$  to block  $H^+(Q_{HP}^{EC})$ .

**Step 4.** Erase block  $H^-(Q_{CP}^{EC})$ .

**Step 4a.**  $S \leftarrow S \cup \{H^-(Q_{CP}^{EC})\}$ .

**Step 5.** Associate block  $H^+(Q_{HP}^{EC})$  with the cold pool, and block  $H^-(Q_{CP}^{EC})$  with the hot pool.

**Step 6.** Reset  $EEC(H^+(Q_{HP}^{EC}))$  and  $EEC(H^-(Q_{CP}^{EC}))$  to zeros.

**Step 6b.** Remove entries of blocks  $H^+(Q_{HP}^{EC})$  and  $H^-(Q_{CP}^{EC})$  from the QH table.

Steps 0a, 4a, and 6b are newly added ones. In Step 0a, spare block  $s_b$  is allocated from  $S$  in a FIFO fashion. Step 1 moves hot data to spare block  $s_b$ , and Step 3 does cold-data migration to cease the wearing of the old block  $H^+(Q_{HP}^{EC})$ . Step 4a joins the now empty block  $H^-(Q_{CP}^{EC})$  to  $S$ , and Step 5 does the hot-cold regulation. Note that, in Step 2, 4, and 6, the wearing information of QH-table entries of blocks  $H^+(Q_{HP}^{EC})$  and  $H^-(Q_{CP}^{EC})$  are revised accordingly. Step 6b removes the two QH-table entries because they are “consumed” by operation DS. It is done by simply marking them protected. On segment check-out, the protected QH-table entries will be merged with the EH table and the on-flash BW table. Basically HPR and CPR are not much changed because they involve only changes to blocks’ pool associations:

**Hot-Pool Resize (HPR):** On the completion of a block erasure, the following condition is checked:

$$EC(H^+(Q_{HP}^{EC})) - EC(H^-(Q_{HP}^{EC})) > 2 \cdot TH.$$

If it holds, then

**Step 1.** Move block  $H^-(Q_{HP}^{EC})$  from the hot pool to the cold pool.

**Step 1a.** Remove the entry of block  $H^-(Q_{HP}^{EC})$  from the QH table.

Note that in Step 1a only the QH-table entry of block  $H^-(Q_{HP}^{EC})$  is removed. That is because HPR does not modify the wearing information of block  $H^+(Q_{HP}^{EC})$ .

**Cold-Pool Resize (CPR):** On the completion of a write request the following condition is checked:

$$EEC(H^+(Q_{CP}^{EEC})) - EEC(H^-(Q_{HP}^{EEC})) > TH.$$

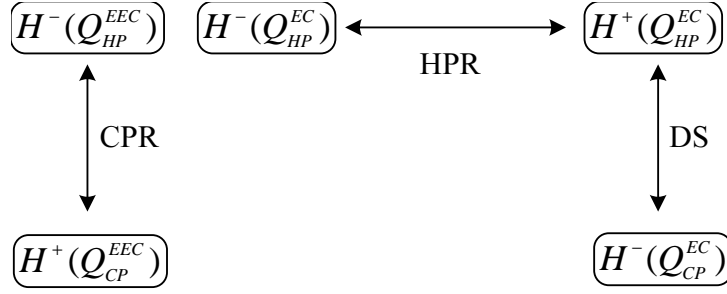


Figure 11: Relations between queue heads and operations DS/CPR/HPR.

If it holds, then

**Step 1.** Move block  $H^+(Q_{CP}^{EEC})$  from the cold pool to the hot pool.

**Step 1a.** Remove the entry of block  $H^+(Q_{CP}^{EEC})$  from the QH table.

Similarly, the QH-table entry of block  $H^-(Q_{HP}^{EEC})$  is left intact.

### 4.3.3 Adaptive Queue-Head Allocation Adjustment

Queue-head entries in the QH table are consumed as they are involved in wear leveling. The relations between queue heads and operations DS/CPR/HPR are shown in Figure 11. For example, one DS operation consumes two queue-head entries from the QH table. So whenever either queue-head entries of  $H^-(Q_{CP}^{EC})$  or  $H^+(Q_{HP}^{EC})$  runs out, DS is inactive until the QH table is refreshed on segment check-out. As CPR and HPR show, some of the queue-head entries are never consumed. So there is no need to reserve more than one entries for the kinds of queue heads. Because only 10 queue-head table entries can be accommodated in RAM, it would then be a question that can they be flexibly allocated among the five queue heads.

An adaptive adjustment algorithm to queue-head allocation is proposed. Whenever an operation is invoked but there is no available queue-head entries can be used, the invocation is failed. Three in-RAM counters, noDS, noCPR, and noHPR referring the numbers of failed invocations to DS, CPR, and HPR, respectively. The adjustment algorithm is periodically invoked. Note that each kind of queue-head must have at least one item. If noDS is found the largest, then we know that queue-head entries of  $H^-(Q_{CP}^{EC})$  or  $H^+(Q_{HP}^{EC})$  should be increased. The two kinds of entries are always incremented in a pair, as they are consumed. The question would then be from which queue head entries may be “stole”. Firstly, as may be notice in

the previous section, only queue-head entries of  $H^-(Q_{HP}^{EEC})$  are never removed. The second choice is to steal one from  $H^-(Q_{HP}^{EC})$  and one from  $H^+(Q_{CP}^{EEC})$ . If both the above two cases fail, then no adjustments are made.

As mentioned above, CPR does not remove queue-head entries of  $H^-(Q_{HP}^{EEC})$ . If noCPR is found the largest, intuitively only queue-head entries of  $H^+(Q_{CP}^{EEC})$  needs to be increased. However, to solely increase queue-head entries of  $H^+(Q_{CP}^{EEC})$  may have no effect. That is because CPR moves a block having large erasure cycles to the hot pool. It subsequently trigger a DS operation to involve the block in cold-data migration. In other words, a CPR operation is always followed by a DS operation. So the adjustment follows two guidelines: 1) To steal queue-head entries from  $H^-(Q_{HP}^{EC})$  (which is used by HPR), and 2) To balance the numbers of queue-head entries of  $H^+(Q_{CP}^{EEC})$  (which is used by CPR),  $H^-(Q_{CP}^{EC})$  (which is used by DS), and  $H^+(Q_{HP}^{EC})$  (which is used by DS).

The case when noHPR is found the largest is similar to the above case. Adjustments can be done analogously. Further details are omitted.

#### 4.3.4 Global Wear Leveling

Because realistic workloads access SSDs with strong localities, some particular segments may receive more writes than other segments. Local wear leveling can not solely level the wearing of all the blocks of flash memory because it is confined to blocks of the same segment. To allow local wear leveling involving blocks of different segment would fundamentally conflict with the segmented management scheme.

To deal with this issue, we propose a global wear-leveling algorithm to conduct wear leveling over segments. Basically a segment is treated as a large unit, in which a local policy takes care of wear leveling over its blocks. As local wear leveling is supposed to be effective, the wearing of a segment is represented by the average block-erasure cycle of all its blocks. Wear leveling over segments is conducted by, again, the dual-pool algorithm. The algorithm needs to be slightly revised, as follows: The erasure cycle and the effective erasure cycle of a segment increase as any of its block is erased. Upon being involved in DS, a segment's effective erasure cycle is reset. Like a block, a segment is associated with either a cold pool or a hot pool. The

	Residence	Size
An erasure-history table	RAM	2*16 Bytes
A queue-head table	RAM	2*60 Bytes
A block-wear table	NAND flash	8*16K Bytes
Local wear-leveling algorithm	NVRAM	1237 Bytes
Global wear-leveling table	RAM	65 Bytes
Global wear-leveling algorithm	NVRAM	997 Bytes
The original firmware	NVRAM	8501 Bytes
Automatic variables	RAM	160 Bytes
Static variables	RAM	289 Bytes
Segment translation table	RAM	2*1000*10 bits

Table 3: Resource utilization of implementing the proposed wear-leveling algorithm.

threshold value  $TH$  is defined in the same way as is that of local wear leveling. However, the effective value equals to  $TH * n$ , where  $n$  is the the number of blocks in a segment. Rationale. One of the challenges of conducting wear leveling over segments is how data can be migrated. We reserved one spare segment for this purpose. Whenever two segments  $s_a$  and  $s_b$  are involved in DS, data are moved from a segment  $s_a$  to the spare segment. All the blocks of segment  $s_a$  is erased, data are moved from segment  $s_b$  to segment  $s_a$ , all the blocks of segment  $s_b$  are erased, and finally segment  $s_b$  is assigned as the spare segment. No doubt the migration of segment data is costly and should not occur frequently. As segment data are migrated, a piece of data is no longer statically mapped to a segment. Thus a small in-RAM translation table is needed to map all the 1000\*16 KB logical-data regions to segments. The mapping table has no more than eight entries and can be quickly constructed on boot.

Because there are only eight segments, a tiny in-RAM table sufficiently accommodates the wearing information of all the segments. The queue heads needed by wear leveling can be extracted by scanning the table. The wearing information of a segment is written as a part of the BW table of the segment. It is committed onto flash memory as a segment is checked out. Note that, as data are migrated among segments, the BW table of segments won't migrate with them.

#### 4.3.5 Resource Utilization

This section provides a summary of resource utilization of implementing the proposed wear leveling algorithm.

Table 3 shows the resource utilization. The in-RAM data structures of local wearing include

two QH tables and two EH tables of two segments. The on-flash data structures are eight BW tables of all the segments. Each BW table occupies a whole block. For global wear leveling, the in-RAM data structures including a small table of segment-wearing information, a small table that maps data regions to segments. The total size is 28 bytes. The segment-wearing information is persistently stored in the BW table of segments. The global wear-leveling algorithm occupies 998 bytes of NVRAM.

The executable code sizes of the local wear-leveling algorithm and the global wear-leveling algorithm are 1237 bytes and 997 bytes, respectively. The total code size is increased by only  $1-(8501+1237+997)/8501=26\%$ .

## 5 Experimental Results

### 5.1 Experimental Setup and Performance Metrics

Our experiments were conducted over an SSD, which is based on the evaluation board FreeScale M68KIT912UF32. The NAND flash on the board is a 128 MB Smart Media card. The page size and the block size are 512 bytes and 16 K bytes, respectively. The mapping-unit size of the SSD firmware is 16 KB. The device is attached to a PC, and the device appears as a 125 MB removable hard drive to the host OS. The volume is formatted in FAT-16. The formatted cluster size is 2 KB.

Instead of using synthesized workloads, realistic workloads are considered in our experiments. A real-life mobile PC was used as the workload generator. An 128 MB disk partition was created on the mobile PC. The file system of the partition is the same as that of the SSD. The host OS is Windows XP. The disk partition is used as a web-browser cache and the storage of an email client. It is to reflect the typical workload of a mobile computer. The disk-block I/O operations were gathered by a filtering driver inserted into the OS kernel. The duration for trace collecting is one week.

The dual-pool algorithm is implemented in the firmware of the SSD. Typical wear-leveling algorithms, algorithm 2L [38], algorithm HC [30, 9], algorithm TB [5] are also implemented and evaluated. Note that comprehensive evaluation and comparison of wear-leveling algorithms can be found in [15]. Our experiments focus on whether the proposed implementation of the



dual-pool algorithm performs close to how the algorithm does in simulation.

To conduct experiments, the gathered traces were replay over the SSD volume **50** times to simulate long-term use. It is carried out by non-buffered low-level I/O APIs (i.e., CreateFile and WriteFile). Two major performance metrics are adopted. The first is the standard deviation of all block-erasure cycles. It reflects how even the distribution all the block-erasure cycles are. The smaller the standard deviation is, the more even the wearing of all the blocks is. The other metric is the overhead ratio. Let the overhead be defined as the summation of all the block-erasure cycles. The overhead ratio denotes the overhead with wear leveling to the overhead without wear leveling. The smaller the overhead ratio is, the less traffic is introduced by wear-leveling activities. Additionally, extra delay imposed by wear-leveling activities on the handling of writes is measured in terms of response time.

## 5.2 Local Wear Leveling

This part of experiments focus on local wear leveling. The gathered traces are replayed over the SSD. Because the NAND flash is partitioned into eight segments, to investigate the performance of local wear leveling, we choose to present the wearing of blocks of the second segment (i.e., block 1024 to block 2047). Global wear leveling is disabled.

Five algorithms are evaluated: algorithm EP (no wear leveling, the original firmware), algorithm HC, algorithm 2L, algorithm TB, and algorithm DP. Algorithm TB is configured with setting 99-1: After every 99 block erasure, a block of all valid data is erased. Algorithm HC, algorithm 2L, and algorithm DP adopt parameter TH, which is the threshold value of the maximum difference among block-erasure cycles to trigger wear leveling. All the three algorithms adopted two TH values 8 and 16. The smaller the TH value, the more aggressive the wear-leveling activities will be. Algorithm HC and algorithm 2L needs another parameter AP, which refers to the number of block erasures between two consecutive invocations to wear leveling. The smaller the AP value, the more frequent wear leveling is invoked. AP is configured as 100 and 200 for TH=8 and TH=16, respectively. in the experiments. Algorithm 2L and algorithm HC need two queue heads, one is the block having the largest erasure cycle and the other one having the smallest erasure cycle. As algorithm DP does, they adopt the in-RAM QH table, the in-RAM EH table, and the on-flash BW table. All the algorithms have



	Max ECs	Min ECs	Total ECs	STDDEV	Overhead Ratios	ECs contributed by		
						BW-table merge	User writes	WL overheads
EP	7,534	0	1,847,150	1505.92	1.00	N/A	1,847,150	N/A
TB,X99Y1	4,058	525	2,308,231	514.24	1.25	442,610	1,847,150	18,471
2L TH=8, AP=100	3,089	2,103	2,278,692	133.55	1.23	413,091	1,847,150	18,451
2L TH=16, AP=200	3,806	1,968	2,268,500	267.35	1.23	412,119	1,847,150	9,231
HC TH=8 AP=100	19,579	2,105	2,298,920	560.42	1.24	414,879	1,847,150	23,339
HC TH=16 AP=200	10,499	1,964	2,278,614	384.47	1.23	413,020	1,847,150	18,444
DP TH=8	2,538	2,506	2,577,420	5.17	1.40	411,200	1,847,150	319,070
DP TH=16	2,384	2,324	2,401,948	10.74	1.30	411,200	1,847,150	143,598

Table 4: Standard deviations and overhead ratios of the third segment under different wear-leveling algorithms.

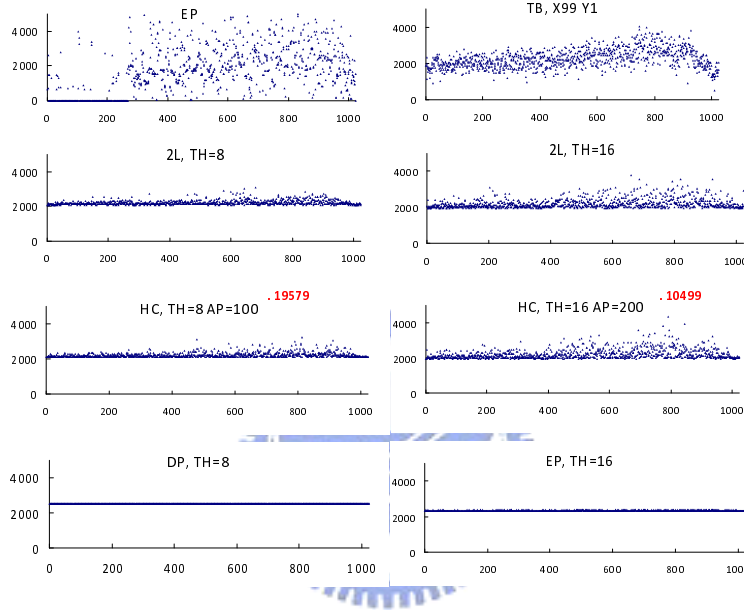
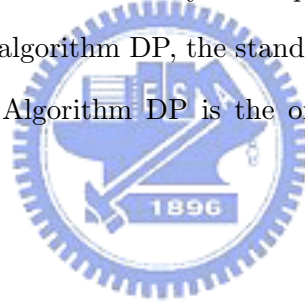


Figure 12: The distribution of erasure cycles of the third segment under different wear-leveling algorithms.

8 entries in the EH table, and 2 entries every different type of queue head in the QH table. The standard deviations and overhead ratios of the algorithms are shown in Table 4. Not surprisingly, the results are consistent with that presented in [15]: Algorithm EP resulted in a very large standard deviation because it ignores blocks having static data residing in. It conducted 1,847,150 block erasures conducted, which serves as the baseline of overhead ratios. Algorithm TB introduced relatively light traffic but its wear leveling is ineffective. Algorithm HC repeatedly involved the oldest block in wear leveling, so the maximum block-erasure cycles are the significantly larger than other algorithms'. Algorithm 2L performed slightly better than the above algorithms. As to algorithm DP, its standard deviations are surprisingly low

(i.e., 5.17 and 10.74). The overhead ratios are comparable to the other algorithm. Table 4 further provides erasure cycles contributed by different activities. Different algorithms' overheads of BW-table merge are close, as algorithm DP spent slightly more efforts on wear-leveling activities. Interestingly, algorithm HC directed almost all the erasures to the oldest block, as the wear-leveling overheads are close to the maximum erasure cycles.

Figure 12 provides the distributions of block-erasure cycles of different local wear-leveling algorithms. Note that, of algorithm HC, the maximum erasure cycles fall beyond the scope of the charts. By observing above results, we two conclusions are reached: 1) The approximation implementation of algorithm 2L and algorithm HC behave similar as they do in simulation presented in [15]. As a crucial building block of wear-leveling algorithms is priority queue, it suggests that the proposed QH table, EH table, and BW table are applicable to the implementation of any other wear-leveling algorithms. 2) The approximation implementation of algorithm DP behaves as expected. The feasibility of the proposed implementation is proven. We found that, except the case of algorithm DP, the standard deviations got larger and larger as experiments were conducted. Algorithm DP is the only algorithm that converged and stabilized its standard deviations.



### 5.3 Global Wear Leveling

In this part of experiments, the global wear-leveling algorithm is evaluated. The proposed local wear-leveling algorithm is enabled to handle blocks in the same blocks. Because a segment is reserved as a spare segment for segment data migration, the device reports itself as a 109 MB removable hard drive. Note that, in this part of experiments, the trace was replayed over the SSD for only 1 time.

Table 5 shows the standard deviations of blocks' erasure cycles of each segment, the standard deviations of erasure cycles of all the blocks on the NAND flash (i.e., row STDDEV Segment 0-7), the total number of erasure operations performed to all the blocks (i.e., row Total EC), and the overhead ratios over all the blocks (i.e., row Overhead ratios). As Table 5 shows, with a small  $TH$  value for global wear leveling, the standard deviations of all the segment blocks are small. The price paid to is relatively large overhead ratios because of segment data migration. Note that, because segment data migration would affect the placement of

Local TH	$\infty$	8	8	8	8	8
Global TH	$\infty$	$\infty$	$2*8*1024$	$8*8*1024$	$16*8*1024$	$32*8*1024$
STDDEV segment 0	63.866	7.681	8.381	7.430	7.681	7.681
STDDEV segment 1	55.147	4.606	6.723	5.915	5.362	4.402
STDDEV sc						5.586
STDDEV sc						15.860
STDDEV sc						19.625
STDDEV sc						15.430
STDDEV sc						10.926
STDDEV sc						5.130
STDDEV sc						108.412
Total EC						1392424
Overhead ratio						1.131

Table 5: Standard deviation of erasure cycles under different settings.

it segments under

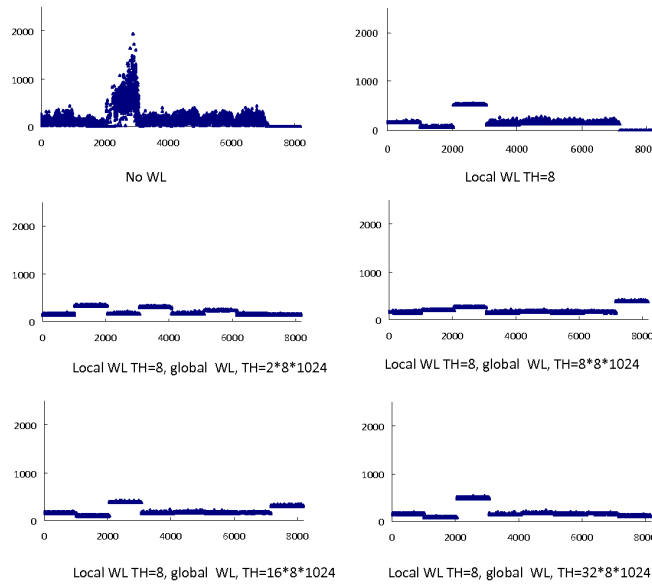


Figure 13: Distribution of block-erasure cycles of all the blocks under global wear leveling.

data inside segment, the behaviors of external wear leveling are not exactly like to shuffle the standard deviations of the segments.

Figure 13 shows the distribution of erasure cycles of all the blocks. It clearly shows that, when the  $TH$  value is either  $2*8*1024$  or  $8*8*1024$ , global wear leveling helps much in balancing the wearing of segment blocks. In consideration of overhead ratios,  $8*8*1024$  will be a good choice.

Because wear-leveling activities may interfere the handling of requests to the SSD, the interference is measured in terms of response time. Response of the handling of writes is particularly interested because reads involve no wear leveling. Table 6 shows the response of writes. It can be seen that, by activating local wear leveling solely, the response is not noticeably affected.

Local TH	$\infty$	8	8
Global TH	$\infty$	$\infty$	8*8*1024
Response maximum	0.218484	1.119857	11.95888
Response minimum	0.001114	0.002164	0.002241
Response average	0.013579	0.015099	0.017898
Response STDDEV	0.013373	0.01466	0.16365

Table 6: Response time of writes in seconds.

When global wear leveling is activated, even though in average the response does not much increased (from 0.017898 to 0.013579), the longest response become lengthy (from 0.218484 to 11.95888). That is caused by segment data migration. User requests may be blocked until a migration completes. We are improving our design to carry out segment data migration in background to alleviate this problem.

## 5.4 Resource-Usage Optimization

The in-RAM data structures required by the proposed implementation are accommodated in a pinch. However, it is worthy to investigate whether significant performance improvement can be achieved by further squeezing more RAM space for the data structures. On the other hand, it is also interesting that how large the in-RAM data structures should be if a more powerful SSD controller is adopted. A simulator based on the firmware code is built to investigate these issues.

### 5.4.1 Simulation validation

A simulator based on the firmware is built so that changes like to enlarge the RAM space can be made. The low-level functions that interact with hardware are replaced by abstraction routines. For example, routines of page reads, page writes, and block erase are replaced by functions of NAND-flash emulation. The DMA module and the USB interface are abstracted accordingly. Basically the firmware can directly be transplanted into the simulator without many modifications.

To ensure that the simulation achieves satisfiable fidelity, validation of the simulator is conducted. Three workloads, a sequential-access workload, a random-access workload, and the gathered traces, are reply over our simulator and the SSD. The final standard deviations and overhead ratios measured from the two platforms are compared, as shown in Table7. The

	The simulator		The real SSD	
	STDDEV	Total ECs	STDDEV	Total ECs
Sequential access	0.6594	112500	0.6954	112500
Random access	11.2425	363420	11.0388	364797
The gathered traces	4.87	1417515	4.9	1419799

Table 7: Simulation validation using different types of workload.

	Static QH Allocation			Adaptive QH Allocation		
	(3,3,3,3,3)	(4,4,4,4,4)	(5,5,5,5,5)	(3,3,3,3,3)	(4,4,4,4,4)	(5,5,5,5,5)
Initial allocation	(3,3,3,3,3)	(4,4,4,4,4)	(5,5,5,5,5)	(3,3,3,3,3)	(4,4,4,4,4)	(5,5,5,5,5)
STDDEV	14.307	12.15	12.925	9.338	10.4211	6.884
Total erasure cycles	176498	178168	178358	178588	178582	180186
Final allocation	(3,3,3,3,3)	(4,4,4,4,4)	(5,5,5,5,5)	(6,1,1,6,1)	(8,2,1,8,1)	(9,5,1,9,1)

Table 8: Static allocation and adaptive allocation of QH table entries among queue heads.

The tuple of allocation stands for the number of queue-head entries of  $(H^+(Q_{HP}^{EC}), H^-(Q_{HP}^{EC}), H^-(Q_{HP}^{EEC}), H^-(Q_{CP}^{EC}), H^+(Q_{CP}^{EEC}))$ .

results show that the differences between the results measured from the simulator and the SSD are negligible under all the three workloads.

#### 5.4.2 QH Table and Adaptive Queue-Head Allocation Adjustment

This part of experiments focus on how wear leveling is affected if the QH-table can be relatively large. Performance of static allocation and dynamic allocation of queue-head entries are also compared.

Table 8 shows the performance of the dual-pool algorithm under static allocation and adaptive allocation of queue-head allocation. The adaptive allocation has been illustrated in Section 4.3.3. It shows that, in general, the more queue-head entries are available, the smaller the standard deviations are. Of course the price paid to is slightly increased total erasure cycles. The other important observation is that in all cases the queue-head entries are better utilized when they are adaptively allocated. The standard deviation is effectively suppressed. By examining the final allocation of queue heads, it can be found that 1)  $H^+(Q_{HP}^{EC})$  and  $H^-(Q_{CP}^{EC})$  have the same number of queue-head entries since DS consumes two each time, 2)  $H^-(Q_{HP}^{EEC})$  always has one entry left because CPR consumes none of its entries, and 3)  $H^-(Q_{HP}^{EC})$  and  $H^+(Q_{CP}^{EEC})$  have relatively small numbers of queue heads since HPR and CPR occur infrequently. The results suggest that, rather than to increase the numbers of queue-head entries,

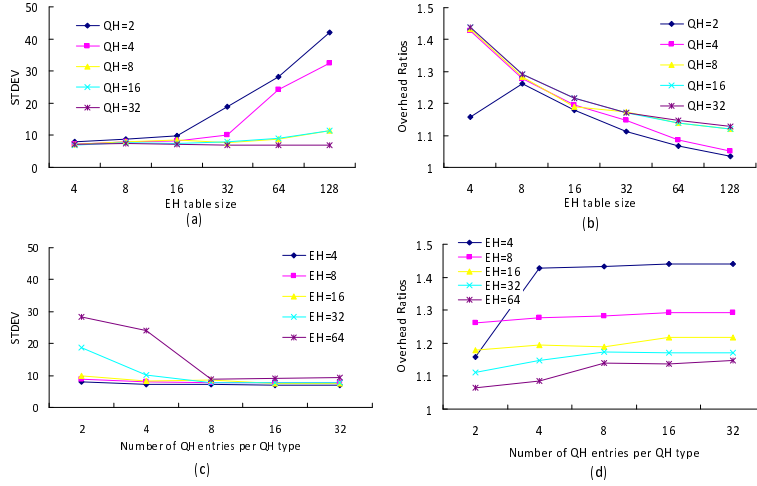


Figure 14: Effects of large EH tables and QH tables.

to adopt adaptive queue-head allocation is more effective.

### 5.4.3 Effects of Large EH tables and QH tables

This section is to examine whether larger EH tables and QH tables would result in better performance. Besides to evaluate whether to further squeeze the current resource utilization is worthy, our intention is to provide insights into the design of an SSD controller.

Simulations were conducted with the same parameters as that in prior experiments, except the sizes of the EH table and the QH table. In our current design, the EH table has 8 entries and the QH table has 2 entries per queue head (algorithm DP uses five queue heads). Figure 14(a) and Figure 14(b) show the standard deviations and overhead ratios under different sizes of the EH table with respect to a particular size of the QH table. For example, “QH=2” refers to that each queue head has two entries, and “EH=8” refers to that there are eight slots in the EH table. Figure 14(a) shows that, the standard deviations becomes large with a large EH table. That is because, as a large amount of erasure history is kept in RAM, algorithm DP may not see the most up-to-date block wearing information because the QH table is not refreshed until BW-table merge takes place. On the other hand, as Figure 14(b) shows, the benefit of using large EH tables is that the overhead ratios are largely reduced because BW-table merge is not frequently invoked.

The effects of using large QH table are indicated by Figure 14(c) and Figure 14(d). Figure

14(c) shows that, interestingly, besides the cases of very large EH tables, to increase the amount of available QH-table entries does not help to reduce standard deviations much. It is the benefit of using the adaptive QH-table entry allocation policy. As to the cases of large EH tables, to provide many QH-table entries happens to cancel the negative effects as mentioned in the last paragraph. Figure 14(c) shows that the overhead ratios are slightly increased as the amount of QH-table entries is large. As algorithm DP has many available QH-table entries to use, the case that algorithm DP is left idle due to insufficient QH-table entries would be infrequent.

## 6 Conclusion

Large and cheap solid-state disks are widely deployed in mobile computers in the recent years. In this paper, we have investigated the feasibility of realizing advanced wear-leveling in a resource-restrictive SSD controller. As wear leveling is concerned with the wearing of each individual block, the challenges mainly pertain to 1) how prioritization over a large number of blocks in terms of their wearing information can be maintained by using limited RAM space, and 2) how block-wearing information can be persistently stored on flash memory without introducing heavy traffic to flash memory. A resource-efficient implementation of a previously proposed wear-leveling algorithm, the dual-pool algorithm, is presented. The implementation requires no more than 200 bytes of RAM space and consumes no noticeable computational power of an SSD controller. It is shown that wear leveling is effectively and efficiently conducted on a large number of blocks by the proposed implementation as if the algorithm had rich computational resources. Besides engineering matters, the contribution of this work is to close the gap between advanced wear-leveling algorithms and SSD implementation.

## References

- [1] A. Kawaguchi, S. Nishioka, and H. Motoda, “A flash-memory based File System,” Proceedings of the USENIX Technical Conference, 1995.
- [2] A. Inoue and D. Wong, “NAND Flash Applications Design Guide,” <http://www.semicon.toshiba.co.jp/eng/prd/memory/doc/pdf/>

nand\_applicationguide\_e.pdf, April, 2003.

- [3] C. Manning and Wookey, "YAFFS Specification," Aleph One Limited, <http://www.aleph1.co.uk/node/37>, Dec, 2001.
- [4] C. Reummler and J. Wilkes, "UNIX disk access patterns," Proceedings of USENIX Technical Conference, 1993.
- [5] D. Woodhouse, "JFFS: The Journalling Flash File System," Proceedings of Ottawa Linux Symposium, 2001.
- [6] E. Gal and S. Toledo, "Algorithms And Data Structures For Flash Memories," ACM Computing Surveys, Vol. 37, Issue 2, 2005.
- [7] F. Douglass, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," Proceedings of the USENIX Operating System Design and Implementation, 1994.
- [8] G. M. Adelson-Velskii and E. M. Landis. "An algorithm for the organization of information," Soviet Math. Doclady 3, pages 1259-1263, 1962.
- [9] H. J. Kim and S. G. Lee, "A New Flash-Memory Management for Flash Storage System," Proceedings of the Computer Software and Applications Conference, 1999. "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," IEICE Transactions on Information and System, Vol. E85-D, No. 6, 2002.
- [10] J. W. Hsieh, T. W. Kuo, and L. P. Chang "Efficient Identification of Hot Data for Flash Memory Storage Systems," ACM Transactions on Storage, Volume 2, Issue 1, 2006.
- [11] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," IEEE Transactions on Consumer Electronics, Vol. 48, No. 2, 2002.
- [12] K. Kim and J. Choi, "Future Outlook of NAND Flash Technology for 40nm Node and Beyond," The 21-st IEEE Non-Volatile Semiconductor Memory Workshop, 2006.



- [13] L. P. Chang, T. W. Kuo, "Real-time Garbage Collection for Flash-Memory Storage System in Embedded Systems," *ACM Transaction on Embedded Computing Systems*, Vol 3, No. 4, 2004.
- [14] L. P. Chang, and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," *Proceedings of The 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.
- [15] L. P. Chang and T. W. Kuo, "An efficient management scheme for large-scale flash-memory storage systems," *Proceedings of the ACM Symposium on Applied Computing*, 2004; "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation", *ACM Transactions on Storage*, Vol. 1, Issue 4, 2005.
- [16] L. P. Chang, "On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems," *Proceedings of the 22st ACM Symposium on Applied Computing*, 2007.
- [17] M. L. Chiang, Paul C. H. Lee, and R. C. Chang, "Using Data Clustering To Improve Cleaning Performance For Flash Memory," *Software - Practice and Experience*, Vol. 29, No. 3, 1999.
- [18] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.
- [19] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, "Flash Memories" *Kluwer Academic Publishers*, 2001.
- [20] R. G. Seidel and C. R. Aragon, "Randomized Search Trees," *Algorithmica*, 16:464-497 (1996).
- [21] T. Gleixner, F. Haverkamp, and A. Bitvutskiy, "UBI - Unsorted Block Images," <http://www.linux-mtd.infradead.org/doc/ubi.html>, 2006.
- [22] W. Vogels, "File system usage in Windows NT 4.0," *Proceedings Of the 17-th Acm Symposium On Operating Systems Principles*, 1999

- [23] Aleph One Company, “Yet Another Flash Filing System”.
- [24] Compact Flash Association, “*CompactFlash<sup>TM</sup>* 1.4 Specification,” 1998.
- [25] Freescale Semiconductor, “RDHCS12UF32TD : USB Thumb Drive Reference Design”
- [26] “USB Thumb Drive reference design DRM061,”Freescale Semiconductor, September, 2004.
- [27] Intel Corporation, “Understanding the Flash Translation Layer(FTL) Specification”.
- [28] “MC9S12UF32 System on a Chip Guide V01.04,”Motorola,inc.,2002
- [29] M-Systems, “Flash-memory Translation Layer for NAND flash (NFTL)”
- [30] M-Systems, “TruFFFS Wear-Leveling Mechanism,” Technical Note TN-DOC-017.
- [31] “Wear Leveling in Single Level Cell NAND Flash Memories,” STMicroelectronics Application Note (AN1822), 2006.
- [32] Samsung Electronics Company, “K9NBG08U5M 4Gb \* 8 Bit NAND Flash Memory Data Sheet”.
- [33] Samsung Electronics Company, “K9GAG08U0M 2G \* 8 Bit NAND Flash Memory Data Sheet (Preliminary)”.
- [34] Samsung Electronics Company, “NAND Flash-based Solid State Disk Data Sheet,” [http://www.samsung.com/ Products/ Semiconductor/ FlashSSD/ download/ Standard\\_type.pdf](http://www.samsung.com/Products/Semiconductor/FlashSSD/download/Standard_type.pdf).
- [35] “K9F2808U0B 16Mb\*8 NAND flash-memory Data Sheet,” Samsung Electronics Company.
- [36] SSFDC Forum, ”*SmartMedia<sup>TM</sup>* Specification”, 1999.
- [37] SanDisk Corporation, “Sandisk Flash Memory Cards Wear Leveling, ” <http://www.sandisk.com/Assets/File/OEM/WhitePapersAndBrochures/RS-MMC/WPaperWearLevelv1.0.pdf>, 2003.

- [38] “Wear Leveling in Single Level Cell NAND Flash Memories,” STMicroelectronics Application Note (AN1822), 2006.

