

# 國立交通大學

## 資訊科學與工程研究所

### 碩士論文

多通道固態硬碟上能自我調整的平均抹除演算法

An Adaptive, Low-Cost Wear-Leveling Algorithm for  
Multichannel Solid-State Disks

研究生：周棟揚

指導教授：張立平 教授

中華民國 101 年 8 月

多通道固態硬碟上能自我調整的平均抹除演算法

An Adaptive, Low-Cost Wear-Leveling Algorithm for Multichannel Solid-State Disks


研究生：周棟揚

Student：Tung-Yang Chou

指導教授：張立平

Advisor：Li-Ping Chang

國立交通大學  
資訊科學與工程研究所  
碩士論文



A Thesis  
Submitted to Department of Computer Science and Engineering  
College of Computer Science  
National Chiao Tung University  
in partial Fulfillment of the Requirements  
for the Degree of  
Master  
in  
Computer Science

August 2012

Hsinchu, Taiwan, Republic of China

中華民國 101 年 8 月

# An Adaptive, Low-Cost Wear-Leveling Algorithm for Multichannel Solid-State Disks

Student : Tung-Yang Chou

Advisor : Li-Ping Chang

Department of Computer and Information Science  
National Chiao Tung University

## Abstract

Multilevel flash memory cells double or even triple storage density, producing affordable solid-state disks for end users. Flash lifetime is becoming a critical issue in the popularity of solid-state disks. Wear-leveling methods can prevent flash-storage devices from prematurely retiring any portions of flash memory. In modern solid-state disks, wear leveling must consider wear evenness at the block level and channel level. In block-level, this study presents an wear-leveling for page-level mapping FTL. Because realistic workloads introduce uneven channel utilizations, block-level wear evenness does not guarantee the maximum device lifetime. This study introduces a channel-level wear-leveling strategy that aims at an eventually-even state of channel lifetimes. A series of trace-driven simulations show that the proposed design outperforms existing approaches in terms of wear evenness and overhead reduction.

## Acknowledgements

I am greatly indebted to my supervisor, Professor Li-Ping Chang, for his patient and valuable instructions on my thesis as well as teaching the right attitude toward work. During my graduate studies, my supervisor encourages me continuously such that I can accomplish this work.

I also owe a special debt of gratitude to all the professors in Computer Science and Engineering Institute, from whose devoted teaching and enlightening lectures I have benefited a lot not only for thesis but also work on the future. Professor Shiao-Li Tsao, you are a wonderful teacher let me into the world of embedded system. Professor Sheau-Ling Hsieh, you have led me profoundly known what network is.

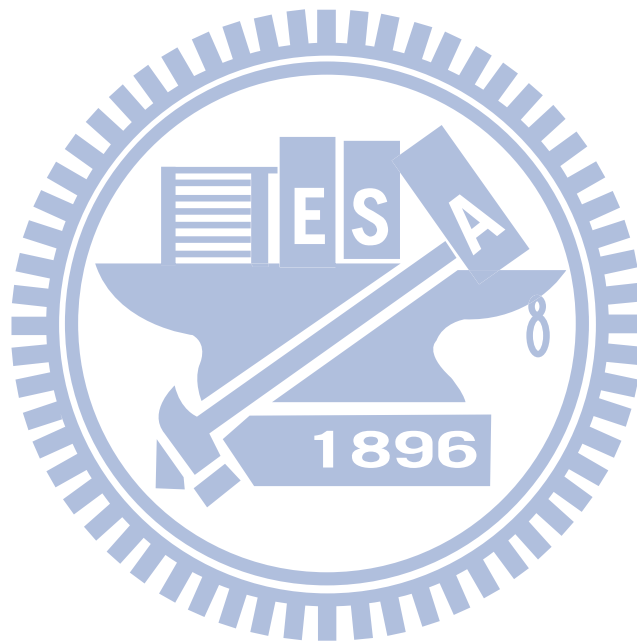
I feel grateful to be a member of ESSLAB. Thanks to those graduated last year, Wen-Hui Lin, Ying-Jie Li, Yi-Cheng Wu and Wei-Han Wang, your selfless sharing experience and kindly treat us as family. Especially thanks to Ying-Jie Li, I spare amount of his working time on company to solve my problem. For Wen-Ping Li, Yi-Kang Chang, Po-Han Sung and Chen-Yi Wen, thanks to your valuable suggestion and critique on my research. For Sheng-Min Huang, Cheng-Yu Hung, Chao-Yuan Mao, Ting-Chieh Huang, thanks for making our lab to alive since senior left.

Last but not least, my gratitude also extend to my family and my lover who have been assisting, supporting and caring for me all of my life without a word of complaint. It is your faith let me pass through every slump in my research. I would never finish without your mentally and materially supporting. Thank you, I love you permanently.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Problem Formulation</b>	<b>3</b>
2.1 Flash Management . . . . .	3
2.1.1 Flash-Memory Characteristics . . . . .	3
2.1.2 Flash Translation Layers (FTLs) . . . . .	4
2.2 The Need for Wear Leveling . . . . .	6
2.2.1 Block-Level Wear Leveling . . . . .	6
2.2.2 Channel-Level Wear Leveling . . . . .	9
<b>3 Block-Level Wear Leveling</b>	<b>10</b>
3.1 Observations . . . . .	10
3.2 The Lazy Wear-Leveling Algorithm for Page-level Mapping . . . . .	11
3.3 Cold Data Selection . . . . .	12
3.3.1 Bitmap . . . . .	12
3.3.2 Least Recently Invalidation List . . . . .	13
3.4 Interacting with Flash-Translation Layers . . . . .	14
<b>4 Channel-Level Wear Leveling</b>	<b>16</b>
4.1 Multichannel Architectures . . . . .	16
4.2 Aligning Channel Lifetime Expectancies . . . . .	18
4.3 Adjusting Channel Utilizations . . . . .	20
<b>5 Performance Evaluation</b>	<b>22</b>
5.1 Experimental Setup and Performance Metrics . . . . .	22

5.2	Experimental Results: Block-Level Wear Leveling . . . . .	23
5.3	Experimental Results: Channel-Level Wear Leveling . . . . .	25
<b>6</b>	<b>Conclusion</b>	<b>27</b>
	<b>Bibliography</b>	<b>27</b>



# List of Figures

1	Two flash-translation layer designs based on hybrid mapping. (a) The set-associative mapping scheme with $N=2$ and $K=2$ . Every group has two logical blocks and a group is allocated to up to two log blocks. (b) The fully-associative mapping scheme. All logical blocks are in one big group and all the log blocks are shared by the logical blocks in this big group. (c) The Page-level mapping scheme. Each logical page has it's own mapping information in RAM. . . . .	5
2	Physical blocks and their erase recency and erase counts. An upward arrow indicates that a block is recently increasing its erase count. . . . .	10
3	Invalidation Bitmap - One bit is for a flash block, and each bit indicates whether a flash block recently receives a page invalidation or not. In cold data selection, it scan the bitmap, and it clears bit of 1's until it encounters a bit of 0. . . . .	13
4	Least Recently Invalidation list - One entry is for a flash block on the GC list, and if there is a page be invalidated, the corresponding entry on the list will be dropped to the end of list.(a) The initial state of LRI. (b) The LRI list after invalidating sequence 2,3,1. . . . .	14
5	Two cases in doing lazy wear leveling for page level mapping. (a) Case1: The <i>lastColdBlock</i> is not null, so choosing cold data is necessary. (b) Case2: The <i>lastColdBlock</i> is null, so continuing copying valid page on <i>lastColdBlock</i> . . . . .	15
6	Handling three write requests $w_1$ , $w_2$ , and $w_3$ using (a) synchronized channels and (b) independent channels. In this example, using synchronized channels doubles the flash wear, while using independent channels results in unbalanced flash wear among channels. . . . .	18
7	Aligning the lifetime expectancies of two channels $C_i$ and $C_j$ for channel-level wear leveling. (a) These two channels reach their end-of-life at different times. (b) Change channel utilizations $u_{c_i}$ and $u_{c_j}$ to $u'_{c_i}$ and $u'_{c_j}$ , respectively, such that the lifetime difference becomes zero (i.e., $d=0$ ). . . .	19
8	Swapping logical blocks among channels for channel wear leveling. (a) Before the swap and (b) after the swap. . . . .	21

9	Final distributions of block erase counts under the PC workload.(a) Not using any wear leveling, (b) using Static wear leveling with TH=12, and (c) using Lazy wear leveling with TH=20. . . . .	24
10	Average erase count on each data written. . . . .	25
11	Runtime channel erase counts and final distributions of block erase counts under the PC workload.(a) Not using channel-level wear leveling, (b) using channel wear leveling with a period length of 512 MB, and (c) using channel wear leveling with a period length of 1 TB. . . . .	25





## List of Tables

1	Comparison of existing algorithms for block-level wear leveling. . . . .	7
2	Symbol definitions. . . . .	19
3	Evaluation results of Lazy wear leveling (LWL) and Static wear leveling (SWL) under the PC workload. “no WL” stands for not using wear leveling.	23



# 1 Introduction

Solid-state disks employ flash memory as their storage medium. The physical characteristics of flash memory differ from those of hard drives, necessitating new methods for data accessing. Solid-state disks hide flash memory from host systems by emulating a collection of logical sectors, allowing systems to switch from a hard drive to a solid-state disk without modifying any existing software and hardware. Solid-state disks are superior to traditional hard drives in terms of shock resistance, energy conservation, random-access performance, and heat dissipation, attracting vendors to deploy such storage devices in laptops, smart phones, and portable media players.

Flash memory is a kind of erase-before-write memory. Because any one part of flash memory can only withstand a limited number of write-erase cycles, approximately 100K cycles under the current technology [18], frequent erase operations can prematurely retire a region in flash memory. This limitation affects the lifetime of solid-state disks in applications such as laptops and desktop PCs, which write disks at very high frequencies. Even worse, recent advances in flash manufacturing technologies exaggerate this lifetime issue. In an attempt to break the entry-cost barrier, modern flash devices now use multi-level cells for double or even triple density. Compared to standard single-level-cell flash, multilevel-cell flash degrades the erase endurance by one or two orders of magnitude [19].

Without wear leveling, localities of data access inevitably degrade wear evenness of flash memory in solid-state disks. Partially wearing out a piece of flash memory not only decreases its total effective capacity, but also increases the frequency of flash erase for free-space management, which further speeds up the wearing out of the rest of the flash memory. A solid-state drive ceases to function when the amount of its worn-out space in flash exceeds what the drive can manage. Wear-leveling techniques ensure that the entire flash wears evenly, postponing the first appearance of a worn-out memory region. However, wear leveling is not free, as it moves data around in flash to prevent solid-state disks from excessively wearing any one part of the memory. As reported in [8], these extra data movements can increase the total number of erase operations by ten percent.

Wear-leveling algorithms include rules defining when data movement is necessary and where the data to move to/from. These rules monitor wear in the entire flash, and intervene when the flash wear develops unbalanced. Wear-leveling algorithms are part of the firmware of solid-state disks, and thus they are subject to crucial resource constraints

of RAM space and execution speeds of solid-state disks' microcontrollers (or simply controller)<sup>1</sup>. Prior research explores various wear-leveling designs under such tight resource budgets, revealing three major design challenges: First, monitoring the entire flash's wear requires considerable time and space overheads, which many controllers in present solid-state disks cannot afford. Second, high implementation complexity discourages firmware programmers from adopting sophisticated algorithms.

Prior methods sort flash erase units in terms of their wear information. This requires efficient access to the wear information of arbitrary erase units, and thus these methods copy the wear information of the entire flash from flash to the RAM of the disk controllers. However, many controllers at the present time cannot afford this RAM space overhead. Chang et al. [5] proposed caching only portions of wear information in RAM. However, the miss penalty and write-back overhead of the cache can scale up the volume of flash-write traffic by up to 10%. Instead of storing the wear information of all flash erase units in RAM, Jung et al. [12] proposed using the average wear of large flash regions. Nevertheless, the low-resolution wear information suffers from distortion whenever flash wearing is severely biased. Chang et al. [8] introduced a bitmap that indicates whether a flash erase unit is recently erased or not. However, using the recent erase history can blind wear-leveling algorithms because the recency and frequency of erasing operations on flash erase units are mutually independent.

From a firmware point of view, implementation complexity primarily involves the applicability of wear-leveling algorithms. The dual-pool algorithm [5] uses five priority queues of wear information and a caching method to reduce the RAM footprints of these queues. The group-based algorithm [12] and the static wear-leveling algorithm [8] add extra data structures to maintain coarse-grained wear information and the recent history of flash wear, respectively. These approaches ignore the information already available in the disk-emulation algorithm, which is a firmware module accompanying wear leveling, and unnecessarily increase their design complexity.

This study presents a wear-leveling algorithm for page-level mapping, based on lazy wear-leveling algorithm [6] to tackle the design challenges mentioned above. Since this design proposed algorithm stores only a RAM-resident bitmap representing pages inval-

---

<sup>1</sup>For example, the GP5086 SSD controller from Global Unichip was rated at 150 MHz and has 64 KB of SRAM for binary executables, data, and mapping tables [10].

idation status in addition to average erase count of the entire flash, achieving a tiny footprint. Second, bitmap structure is easy to maintain, reducing the implementation complexity.

Modern solid-state disks equip with multiple channels for parallel flash operations. In this study, a channel refers to a logical unit that independently processes flash commands and transfers data. Multichannel designs boost the write throughput but introduce unbalanced wear of flash erase units among channels. Prior work address this issue by dispatching write requests to channels on a page-by-page basis [7, 9] (a page is the smallest read/write unit of flash). Dispatching data at the page level requires page-level mapping, whose implementation requires considerable RAM space for large flash. Additionally, this approach could map logically consecutive data to the same channel and degrade the channel-level parallelism in sequential read requests. This study introduces a novel channel-level wear leveling strategy based on the concept of reaching “eventually even” channel lifetimes. The basic idea is to align channels’ lifetime expectancies by re-mapping data among channels. The proposed approach has many benefits, including 1) it does not require a channel-level threshold for wear leveling, 2) it incurs very limited overhead, and 3) it requires only a small RAM-resident data structure.

In summary, this study has the following contributions:

1. An efficient block wear-leveling algorithm with a tiny RAM footprint.
2. An algorithm for wear leveling at the channel level.

The rest of this paper is organized as follows: Section 2 reviews flash characteristics and prior work on flash translation and wear leveling. Section 3 presents an block-level wear-leveling algorithm. Section 4 introduces a strategy for wear leveling at the channel level. Section 5 reports our simulation results. Section 6 concludes this paper.

## 2 Problem Formulation

### 2.1 Flash Management

#### 2.1.1 Flash-Memory Characteristics

Solid-state disks use NAND flash memory (flash memory for short) as their storage medium. A piece of flash memory is a physical array of *blocks*, and each block contains

the same number of *pages*. Typically a flash page is of 2048 plus 64 bytes. The 2048-byte portion stores user data, while the 64 bytes is a spare area for mapping information, block aging information, error-correcting code, etc. Flash memory reads and writes in terms of pages, and overwriting a page requires erasing. Flash erases in terms of blocks, each of which consists of 64 pages. Under the current technology, a flash block can only sustain a limited number of write-erase cycles before it becomes unreliable. A single-level-cell flash block endures 100K cycles [18], while this limit is 10K or less in multilevel-cell flash [19].

Solid-state disks emulate disk geometry using a firmware layer called the flash-translation layer (FTL). FTLs update existing data out of place and invalidate old copies of the data to avoid erasing a flash block every time before rewriting a piece of data. Thus, FTLs require a mapping scheme to translate disk sector numbers into physical flash addresses. Updating data out of place consumes free space in flash, and FTLs must recycle flash space occupied by invalid data with erase operations. Before erasing a block, FTLs copy all valid data from this block to other free space. *Garbage collection* refers to a series of copy and erase operations for reclaiming free space.

### 2.1.2 Flash Translation Layers (FTLs)

Flash-translation layers are part of the firmware in solid-state disks. They use RAM-resident index structures to translate logical page numbers into physical flash locations. Mapping resolutions have direct impact on RAM-space requirements and write performance. Many entry-level flash-storage devices like USB thumb drives adopt block-level mapping, which requires only small mapping structures. However, low-resolution mapping suffers from slow response when servicing small write requests. Page-level mapping [11] better handles random write requests, but requires large mapping structures, making its implementation difficult when flash capacity is high. This paper considers logical pages as the smallest mapping unit as large as a flash page.

Hybrid mapping combines both page and block mapping. This method groups consecutive logical pages into logical blocks as large as physical blocks. It maps logical blocks to physical blocks on a one-to-one basis using a *block-mapping table*. If a physical block is mapped to a logical block, then this physical block is called the *data block* of this logical block. Initially, physical blocks other than data blocks are *spare blocks*. Hybrid mapping uses spare blocks as *log blocks* to serve page updates, and uses a *page mapping table* to

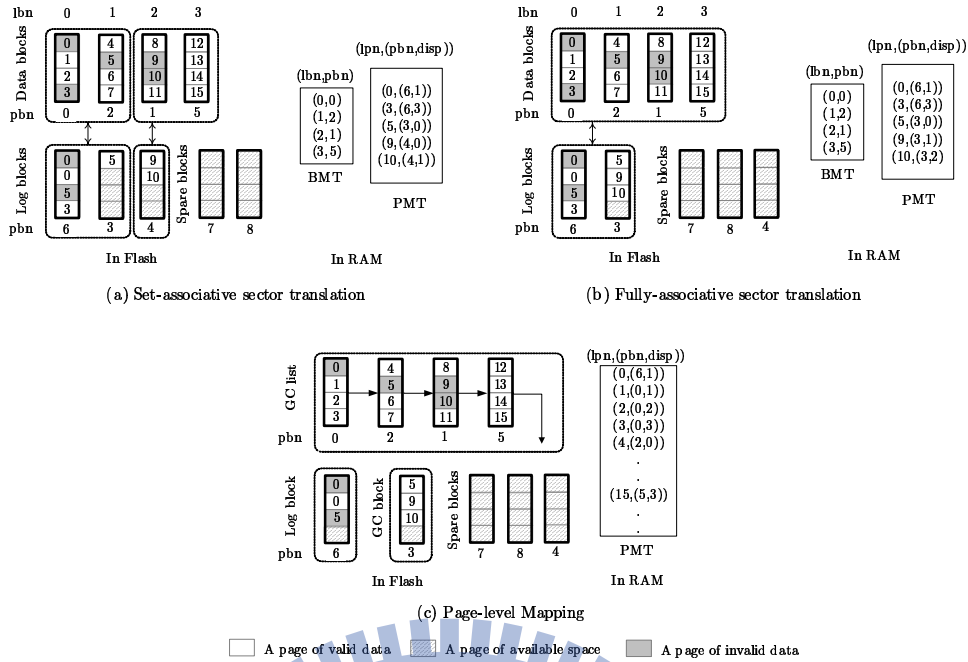


Figure 1: Two flash-translation layer designs based on hybrid mapping. (a) The set-associative mapping scheme with  $N=2$  and  $K=2$ . Every group has two logical blocks and a group is allocated to up to two log blocks. (b) The fully-associative mapping scheme. All logical blocks are in one big group and all the log blocks are shared by the logical blocks in this big group. (c) The Page-level mapping scheme. Each logical page has its own mapping information in RAM.

redirect read requests to the latest versions of data in the log blocks.

Figures 1(a) and 1(b) show two different FTL designs using hybrid mapping, and Figure 1(c) shows a FTL design using Page-level mapping. Hybrid mapping creates groups of logical blocks and allocate (flash) spare blocks as log blocks for these logical-block groups. Let  $lbn$  and  $pbn$  stand for a logical-block number and a physical-block number, respectively. Let  $lpn$  represent a logical-page number, and let  $disp$  be the block offset in terms of pages. The bold boxes stand for physical blocks, each of which has four pages. The numbers in the pages indicate the  $lpns$  of their storage data. The BMT and the PMT are the block-mapping table and the page-mapping table, respectively. In Fig. 1(a), every group has two logical blocks, while a group can be allocated to up to two log blocks. This mapping scheme, developed by Park et al. [16], is called set-associative mapping (SAST). This scheme uses two parameters  $N$  and  $K$  to specify the group size and the largest number of log blocks that a group can have, respectively. Figure 1(b) depicts another mapping scheme, developed by Lee et al. [14], called fully-associative

mapping (FAST). This method put all logical blocks in one big group, and has all the logical blocks in this big group share all the log blocks. Page-level mapping in Fig. 1(c) create a GC-list to link all flash block was filled up. This mapping scheme, developed by Gupta et al. [11]. This scheme maintains a log block to store all updates, and a GC block to reserve the valid page within garbage collection.

The FTL consumes spare blocks for serving incoming write requests. When the amount of spare blocks becomes low, the FTL starts erasing log blocks. Before erasing a log block, the FTL finds all logical blocks related to the valid data in this log block. For each of the found logical block, the FTL collects valid data from the log block and the data block of this logical block, copies these valid data to a new spare block, and re-maps the logical block to the copy-destination spare block. Finally, the FTL erases all the involved data blocks and the log blocks into spare blocks. This procedure is referred to as merge operations or garbage collection. For example, in Fig. 1(a), for garbage collection the FTL collects the valid data scattered in the data blocks at *pbns* 0 and 2 and in the log blocks at *pbns* 6 and 3, write them to the spare blocks at *pbns* 7 and 8, and then erases the four old flash blocks at *pbns* 0, 2, 6, and 3 into spare blocks. In Page-level mapping FTL, the physical block having fewest valid page in GC-list will be selected to do garbage collection.

Hybrid mapping FTLs exhibit some common behaviors in the garbage-collection process regardless of their designs, i.e., garbage collection never involves a data block if none of its page data have been updated. In Fig. 1(a), erasing the data blocks at *pbn* 5 cannot reclaim any free space. Similarly, in Fig. 1(b), erasing any of the log blocks does not involve the data block at *pbn* 5. This is a potential cause of uneven flash wear. This situation same as in Page-level mapping FTL.

## 2.2 The Need for Wear Leveling

This section first introduces prior methods, discusses their drawbacks, and then point out how the method to be proposed improves upon these shortcomings.

### 2.2.1 Block-Level Wear Leveling

Block-level wear leveling considers the wear evenness of a collection of flash blocks. Let the *erase count* of a flash block denote how many write-erase cycles this block has



Algorithm	Principle	RAM-resident data structures required	Threshold tuning
Static wear leveling [8]	Static wear leveling	A block erase bitmap	Manual
Group wear leveling [12]	Hot-cold swapping	Average erase counts of block groups	Manual
Dual-pool wear leveling [5]	Cold-data migration	All blocks' erase counts and their recent erase counts	Manual
Remaining-lifetime leveling [2]	Cold-data migration	All blocks' age information (remaining lifetimes) and block-data temperature (update frequencies)	Manual
Lazy wear leveling (this study)	Cold-data migration	An average erase count of all blocks	Automatic

Table 1: Comparison of existing algorithms for block-level wear leveling.

undergone. There have been three representative techniques for this problem: Static wear leveling, Hot-cold swapping, and Cold-data migration. Static wear leveling moves static/immutable data away from lesser worn flash blocks, encouraging the flash-translation layer to start erasing these blocks. Flash vendors including Micron [15] and Spansion [22] recommend using this approach. Chang et al. [8] described a design of Static wear leveling. However, Chang and Du [5] found Static wear leveling failed to achieve even block wear on the long-term, because Static wear leveling could 1) move static/immutable data back and forth among lesser worn blocks and 2) erase a flash block even if its erase count is relatively large. Hot-cold swapping exchanges data in a lesser worn block with data from a badly worn block. Jung et al. [12] presented a hot-cold swapping design. However, because the oldest block has a very large (and perhaps still the largest) erase count, Chang and Du [5] found that Hot-cold swapping risks erasing the most worn flash block pathologically.

Cold-data migration relocates infrequently updated data (i.e., cold data) to excessively worn blocks to protect these blocks against garbage collection. Preventing badly-worn blocks from aging further is not equal to increasing the wear of lesser-worn blocks



(as Static wear leveling does). This is because frequently updated data occupy only a small portion of the disk space. Prior work reported that the disk fullness of productive systems was only about forty percent [1]. In other words, to stop aging the small amount of badly-worn flash blocks mapped to frequently updated data is more efficient than to start wearing the large amount of lesser-worn flash blocks. Cold-data migration has been proven more effective than Static wear leveling and Hot-cold swapping [2, 5]. Based on Cold-data migration, Argrawal et al. [2] proposed storing the remaining lifetimes and data temperatures of all flash blocks in RAM, and Chang and Du [5] proposed storing all blocks' erase counts and their recent erase counts in RAM. These designs, however, impose large RAM-space requirements on disk controllers. Consider a 32 GB flash-storage device with 512 KB flash blocks, storing a four-byte wear information for every block costs the disk controller 256 KB of RAM. This figure is higher than that a typical disk controller can afford (64 KB, mentioned in the Introduction section). Reducing the RAM footprint is always beneficial no matter how much RAM the controller can afford, because the saved RAM space can be used by the mapping tables and the disk write buffer. Table 1 is a summary of comparison among prior methods and our algorithm. Our design stores only an average erase count in RAM, achieving a tiny RAM footprint. However, our design does not sacrifice wear-leveling performance to footprint reduction. Our experimental results will show that it outperforms existing methods in almost all cases.

Block-level wear leveling controls the wear variance in all flash blocks within an acceptable threshold. Existing approaches have different definitions of this variance: Chang et al. [8] adopted the ratio of the total erase count to the total number of the recently erased blocks, Jung et al. [12] and Chang and Du [5] used the difference among blocks' erase counts, and Argrawal et al. [2] employed the difference among blocks' remaining lifetimes. With a smaller threshold, wear leveling aims at a more level wear in flash blocks, but inevitably introduces more frequent data movement. Wear leveling overhead can be affected by many conditions of flash management, including the host workload, flash-translation layer, flash geometry, and flash capacity.

Unfortunately, it is almost impossible to find a universally applicable threshold setting for various applications of flash storage. For example, in our two tests with Dual-pool algorithm [5] with a threshold of 14, under the workloads of a multimedia appliance and a Windows desktop, it increased the total erase count by 0.8% and 3.9% while the resultant

standard deviations of all blocks' erase counts were 5.4 and 10.5, respectively<sup>1</sup>. The latter case shows that the same threshold setting resulted more data movement but not achieved a better wear evenness. This study identifies that the overhead of wear leveling is not linearly related to the threshold value, and the overhead will significantly increase when the threshold is becoming smaller than a certain critical value. This critical threshold value will be different for various conditions of flash management. Thus, we propose subjecting the threshold value to the overhead increase ratio, and introduce a runtime strategy that dynamically sets the threshold value to the critical value.

### 2.2.2 Channel-Level Wear Leveling

In this study, a channel refers to a logical unit that independently processes flash commands and transfers data. Channel-level wear leveling is concerned with the wear evenness of flash blocks from different channels. This issue is closely related channel binding of logical pages, i.e., the allocation of free flash pages to host data. Dynamic channel binding globally manages free pages across all channels. Chang and Kuo [7] proposed dispatching page write requests to channels based on the update frequencies of these page data. Dirik et al. [9] proposed allocating channels to incoming page write requests using the round-robin policy. Even though dynamic channel binding has better flexibility of balancing the block wear across all channels, it has two drawbacks: 1) it adds extra channel-level mapping information to every logical page, resulting in larger mapping tables and 2) it could map consecutive logical pages to the same channel, severely degrading the channel-level parallelism in sequential-read requests.

Instead of dynamic channel binding, this study considers static channel binding. Static channel binding uses fixed mapping between logical pages and channels. With static mapping, effectively every channel manages its free flash pages with its own instance of flash-translation layer. The most common strategy for static channel binding is the RAID-0-style striping [2, 17, 20]. RAID-0 striping achieves the maximum channel-level parallelism in sequential read because it maps a collection of consecutive logical pages to the largest number of channels. We must point out that RAID-0 striping cannot automatically achieve wear leveling at the channel level. This is because, as reported in [4], hot data (frequently updated data) are small, usually between 4 KB and 16 KB.

---

<sup>1</sup>These disk workloads were used in our experiments. See Section 5.1.

RAID-0 striping statically binds small and hot data to some particular channels, resulting in imbalanced write traffics among channels. We found that, under the disk workload of a Windows desktop, a four-channel architecture had a largest and a smallest fractions of channel-write traffic of 28% and 23%, respectively. Thus, flash blocks from different channels wear at different rates. Extending the scope of block-level wear leveling to the entire storage device is not a feasible solution here, because it requires dynamic channel binding.

### 3 Block-Level Wear Leveling

This section presents an algorithm for wear leveling at the block level. This algorithm does not deal with channels so logically all flash blocks are in the same channel.

#### 3.1 Observations

This section defines some key terms for the purpose of presenting our wear-leveling algorithm in later sections. Let the *update recency* of a logical block denote the time length between the current time and the latest update to this logical block. The update recency of a logical block is high if its latest update is more recent than the average update recency. Otherwise, its update recency is low. Analogously, let the *erase recency* of a physical block be the time length since the latest erase operation on this block. Thus, immediately after garbage collection erases a physical block, this block has the highest erase recency. A physical block is a *senior block* if its erase count is larger than the average erase count. Otherwise, it is a *junior block*.

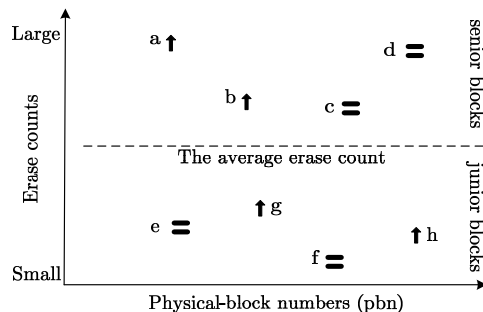


Figure 2: Physical blocks and their erase recency and erase counts. An upward arrow indicates that a block is recently increasing its erase count.

Temporal localities of updating logical blocks affect the wear of physical blocks. As previously mentioned, if a physical block is mapped to an unmodified logical block, then garbage collection will avoid erasing this physical block. On the other hand, updates to logical blocks produce invalid data in flash blocks, and thus physical blocks mapped to recently modified logical blocks are good candidates for garbage collection. After a physical block is erased by garbage collection, it either serves a data block or a log block. Either way, this physical block is again related to recently modified logical blocks. So if a physical block has a high erase recency, then it will quickly accumulate many erase counts. Conversely, physical blocks lose momentum in increasing their erase counts if they are mapped to logical blocks having low update recency.

Figure 2 provides an example of eight physical blocks' erase recency and erase counts. Upward arrows mark physical blocks recently increasing their erase counts, while an equal sign indicates otherwise. Block *a* is a senior block with a high erase recency, while block *d* is a senior block but has a low erase recency. The junior block *h* has a high erase recency, while the erase recency of the junior block *e* is low. Blocks should keep their erase counts close to the average. Two kinds of block wear can require intervention from wear leveling. First, the junior blocks *e* and *f* have not recently increased their erase counts. As their erase counts fall below the average, wear leveling has them start participating in garbage collection. Second, the senior blocks *a* and *b* are still increasing their erase counts. Wear leveling has garbage collection stop further wear in these two senior blocks.

### 3.2 The Lazy Wear-Leveling Algorithm for Page-level Mapping

First, we introduce Lazy Wear-Leveling for Hybrid mapping, and then modify some part of algorithm to adapt in Page-Level mapping. Principle of Lazy Wear-Leveling: whenever a senior block's erase recency becomes high, it will re-locate (i.e., re-map) a logical block having a low update recency to this senior block.

Lazy wear leveling must be aware of the recent wear of all senior blocks, because senior blocks retire before junior blocks. However, physical blocks boost their erase recency only via garbage collection. The flash-translation layer can notify Lazy wear leveling of its decision on victim selection. This way, Lazy wear leveling captures senior blocks whenever their erase recency become high without repeatedly checking all senior blocks' wear information.

How to prevent senior blocks from further aging is closely related to the behaviors of garbage collection. It is impossible to use the same method for hybrid mapping to choose cold data as described in [6], since physical pages of a logical block are scattered in lots of physical blocks in page-level mapping. Therefore, it is hard to select a cold physical block from the original information; instead, we use some additional structure to tackle data temperature.

To re-map a logical block from one physical block to another, Lazy wear leveling moves all valid data from the source physical block to the destination physical block. Junior blocks are the most common kind of source blocks, e.g., blocks  $e$  and  $f$  in Fig. 2, because storing immutable data keeps them away from garbage collection. As moving all valid data out of the source blocks makes them good candidates for garbage collection, selecting logical blocks for re-mapping is related to the wear of junior blocks. To give junior blocks even chances of wear, it is important to uniformly visit every logical block when selecting logical blocks for re-mapping.

Temporal localities of write change occasionally. New updates to a logical block can neutralize the latest re-mapping effort involving this logical block. In this case, Lazy wear leveling will be notified that a senior block is again selected as a victim of garbage collection, and will perform another re-mapping operation for this senior block.

In page-level mapping Lazy wear leveling copies data having low update recency to senior blocks to prevent these blocks from aging further. However, the method that choose a cold data according to mapping info for hybrid mapping proposed in [6] is not available for page-level mapping.

### 3.3 Cold Data Selection

This section presents two algorithm for cold data selection. The first is using a bitmap structure to identify data temperature, and the second is an improvement upon cold data selection, we called LRI list.

#### 3.3.1 Bitmap

This study proposes using an invalidation bitmap. In Fig.3 is an example of Invalidation bitmap, one bit is for a flash block, and each bit indicates whether a flash block recently receives a page invalidation (i.e., 1) or not (i.e., 0). All the bits are 0 initially,

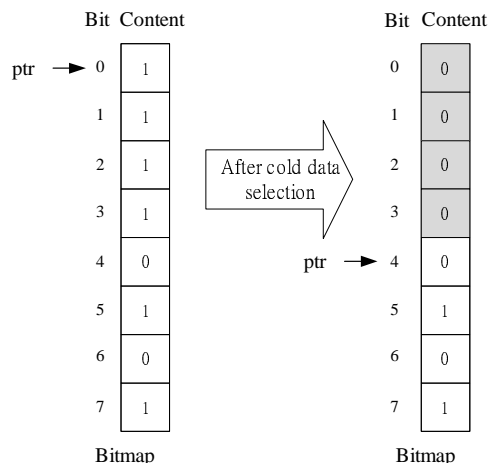


Figure 3: Invalidation Bitmap - One bit is for a flash block, and each bit indicates whether a flash block recently receives a page invalidation or not. In cold data selection, it scan the bitmap, and it clears bit of 1's until it encounters a bit of 0.

and there is a pointer referring to the first bit. The bit of a flash block switches to 1 if any page in this block is updated (i.e., invalidated). Whenever Lazy wear leveling finds the erase count of a victim block larger than the average by  $\Delta$ , it advances the pointer and scans the bitmap. As the pointer advances, it clears bits of 1's until it encounters a bit of 0 as shown in Fig.3. Lazy wear leveling then select the flash block owning this zero bit as cold block. Notice that garbage-collection activities do not alter any bits in the bitmap.

The rationale behind the design is that, in the presence of temporal localities of write, if a flash block does not receive page invalidations recently, then this block is unlikely to receive more page invalidations in the near future. The invalidation bitmap resides in RAM, and it requires one bit per flash block. Compared to the page-level mapping table, the space overhead of this bitmap is very limited.

### 3.3.2 Least Recently Invalidation List

Although Invalidation Bitmap can identify cold data from hot data, but it can't sufficiently differentiate blocks storing cold data from not-recently invalidated block. Because we have no idea of the invalidation sequence from the Invalidation Bitmap, the bit's of 1 in Bitmap just notify that there is a invalidation on the block recently. Therefore, we can't pick out the coldest one from the block owning zero bit in bitmap.

We proposed an optimization upon cold data selection to overcome this problem, it

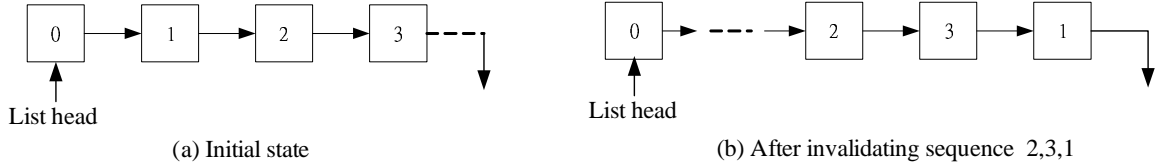


Figure 4: Least Recently Invalidation list - One entry is for a flash block on the GC list, and if there is a page be invalidated, the corresponding entry on the list will be dropped to the end of list.(a) The initial state of LRI. (b) The LRI list after invalidating block sequence 2,3,1, and each block invalidate one page.

is called Least Recently Invalidated(LRI) list. One entry is for a flash block on the GC list, and these entry is linked in sequence initially as shown in Fig.4(a). When there is a page of a block being invalidate, the corresponding entry on the list will be dropped to the end of list. Obviously, list head of LRI list is the coldest block, and this block will be selected to do page copy.

LRI list can pick up a cold block more precisely than Invalidation Bitmap. For example, if there is a block invalidation sequence "2,3,1", it means that there are 3 pages being update in block 2,3 and 1 respectively. In case of Invalidation Bitmap, bits of block 1,2 and 3 will be set to 1 after invalidation. However, it's impossible to distinguish the data temperature between block 1,2 and 3. In contrast, it is easy to extract the coldest one from block 1,2 and 3 after invalidation on LRI list as show in Fig.4(b).

### 3.4 Interacting with Flash-Translation Layers

This section describes how Lazy wear leveling interacts with its accompanying firmware module, the flash-translation layer. Algorithm 1 shows the pseudo code of Lazy wear leveling for Page-Level mapping. The flash-translation layer calls Algorithm 1 after it moves all valid data out of a garbage-collection victim block and before it erases this block. The input of Algorithm 1 is  $v$ , the  $pbn$  of the victim block. This algorithm performs re-mapping whenever necessary, and may impact over than three blocks.

For the example of Page-level mapping in Fig. 1(c), suppose that the flash-translation layer decides to make a garbage collection at  $pbn$  1. The flash-translation layer calls Algorithm 1 before erasing  $pbn$  1. In the rest of this section, concept of our algorithm will be introduced before detail explanation of Algorithm 1

Lazy wear leveling is separated into two cases by detecting the value of *lastColdBlock*.



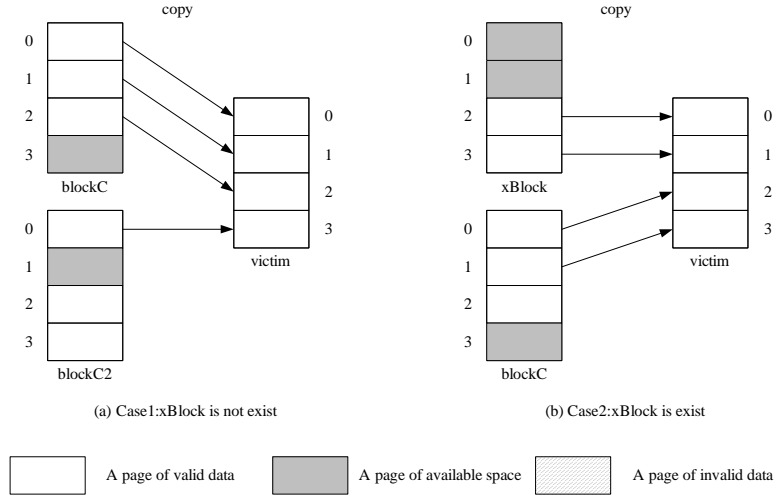


Figure 5: Two cases in doing lazy wear leveling for page level mapping. (a) Case1: The *lastColdBlock* is not null, so choosing cold data is necessary. (b) Case2: The *lastColdBlock* is null, so continue copying valid page on *lastColdBlock*.

The *lastColdBlock* is the block that selected as a cold block in the last lazy wear leveling and there are valid pages left in the cold block in the end of last wear leveling. Initially, *lastColdBlock* is null, and it will be set to null if there is a page be invalidated or the block be erased before next wear leveling occur. In Fig. 5, *blockC* and *blockC2* are cold block that selected by cold data selection algorithm we proposed in section 3.3.1 or 3.3.2. Victim is the block which selected to do garbage collection by flash-translation layer and trigger wear leveling. In case1 Fig. 5(a), *lastColdBlock* is not null, thus we use cold data selection algorithm to select a cold data block *C*. All of the valid pages of *blockC* will be copied to victim one by one, until fill out free pages of victim. If valid pages of *blockC* are exhausted, wear leveling will repeatedly use cold data selection algorithm again to choose a cold block referred as *blockC2*, and then copy valid pages of *blockC2* to fill out the left free pages in victim. When all the free pages of victim are satisfied, page copying will be stopped immediately. If page copying is stopped, and there are still some valid pages left in *blockC2*, then *lastColdBlock* will be marked as *blockC2*. In case2 Fig. 5(b), *lastColdBlock* is null, then page copying from *lastColdBlock* to victim will begin immediately. If *lastColdBlock* has no valid page, wear leveling will use cold data selection algorithm again to choose a cold block referred as *blockC2*. And the rest of progress is same as in case1. The rest of this section is a detailed explanation of Algorithm 1.

In Algorithm 1, the flash-translation layer provides the subroutines with leading un-



derscores, and wear leveling implements the rest. In Step 1, *eraseCount()* obtains the erase count  $e_v$  of the victim block  $v$  by reading the victim block’s page spare area, in which the flash-translation layer stores the erase count. Step 2 compares  $e_v$  against the average erase count  $e_{avg}$ . If  $e_v$  is larger than  $e_{avg}$  by a predefined threshold  $\Delta$ , then Steps 3 through 22 will carry out a re-mapping operation. Otherwise, Steps 24 through 26 process the routine erase. Step 4 check whether *lastColdBlock* is null or not. If *lastColdBlock* is not null, *coldBlock* is assigned as *lastColdBlock* in step 5. Otherwise, *\_findColdBlock* routine that we proposed in section 3.2.1 or 3.2.2 is called to find a cold block and assign to *coldBlock* in step 7. The loop of Steps 9 through 12 repeatedly copy valid page from *coldBlock* to the free page of victim block until *coldBlock* has no valid page or no more than one free page on victim block by using *\_pbnHasValidPages()* and *\_pbnHasFreePage()* two sub-routines. Step 10 uses subroutine *\_ppnNextOfBlock(coldBlock)* to get the next available page of coldBlock to  $p$ . Then step 11 calls subroutine *\_copy()* and *\_map()* to copy the valid page on *coldBlock* to victim, and modify mapping table respectively.

Step 14 uses the subroutine *\_pbnHasFreePage()* to check whether victim has free space or not. If victim has at least one free page, wear leveling will not stop. It means that we have to go back to step 7 to get another *coldBlock*, and page copy will continue until fill out victim block. Step 14 set *lastColdBlock* to null, and go back to step 7 in step 15. Otherwise, the page copy is stop, and step 16 through 22 do post work of wear leveling. In step 16 uses subroutine *\_pbnHasValidPage()* to check whether there are any valid page on *coldBlock*, If there are at least one valid page on *coldBlock*, *lastColdBlock* will be set to *coldBlock* in step 17, if not, *lastColdBlock* will be set to null. After that , Step 9 increases  $e_v$ , since the former victim block  $v$  has been erased, and Step 10 updates the average erase count.

## 4 Channel-Level Wear Leveling

### 4.1 Multichannel Architectures

Advanced solid-state disks use multichannel architectures for high data transfer rates [2, 13, 20, 17]. In this study, a channel stands for a logical unit which can individually handle flash commands and perform data transfer. Parallel hardware structures such as gangs, interleaving groups, and flash planes are part of channels because flash chips in

---

**Algorithm 1** The lazy wear-leveling algorithm

---

**Input:**  $v$ : the victim block for garbage collection

```
1:  $e_v \leftarrow \text{eraseCount}(v)$ 
2: if  $(e_v - e_{avg}) > \Delta$  then
3:    $\_erase(v)$ ;
4:   if  $\text{lastColdBlock} \neq \text{null}$  then
5:      $\text{coldBlock} \leftarrow \text{lastColdBlock}$ 
6:   else
7:      $\text{coldBlock} \leftarrow \_findColdBlock()$ 
8:   end if
9:   while  $\_pbnHasValidPage(\text{coldBlock}) = \text{TRUE}$  AND  $\_pbnHasFreePage(v) = \text{TRUE}$  do
10:     $p \leftarrow \_ppnNextOfBlock(\text{coldBlock})$ 
11:     $\_copy(v, p)$ ;  $\_map(v, p)$ 
12:   end while
13:   if  $\_pbnHasFreePage(v) = \text{TRUE}$  then
14:      $\text{lastColdBlock} \leftarrow \text{null}$ 
15:     go to 7
16:   else if  $\_pbnHasValidPage(v) = \text{TRUE}$  then
17:      $\text{lastColdBlock} \leftarrow \text{coldBlock}$ 
18:   else
19:      $\text{lastColdBlock} \leftarrow \text{null}$ 
20:   end if
21:    $e_v \leftarrow e_v + 1$ 
22:    $e_{avg} \leftarrow \text{updateAverage}(e_{avg}, e_v)$ 
23: else
24:    $\_erase(v)$ ;
25:    $e_v \leftarrow e_v + 1$ 
26:    $e_{avg} \leftarrow \text{updateAverage}(e_{avg}, e_v)$ 
27: end if
```

---

these structures might not be individually programmable.

From the point of view of wear leveling, channels can be *synchronized* or *independent*. Figure 6 is an example. Let the mapping between logical pages and channels use the RAID-0 style striping. Figure 6(a) depicts that all the channels write synchronously even if a write request do not access all the channels. Lazy wear leveling directly applies to a set of synchronized channels because these channels are logically equivalent to a single channel. A major drawback of synching channel operations is the reduced device lifetime. As Figure 6(a) shows, the channels writes sixteen flash pages to modify only eight logical pages. Independent channels need not copy unmodified data for synching channel operations, as shown in Figure 6(b). However, using independent channels inevitably introduces unbalanced flash wear among channels.

This study focuses on independent channels because they alleviate the pressure of garbage collection and reduce flash wear compared to synchronized channels. Let every

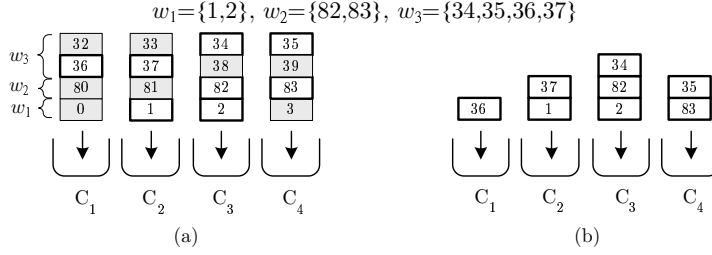


Figure 6: Handling three write requests  $w_1$ ,  $w_2$ , and  $w_3$  using (a) synchronized channels and (b) independent channels. In this example, using synchronized channels doubles the flash wear, while using independent channels results in unbalanced flash wear among channels.

independent channel adopt an instance of flash-translation layer, and let every channel perform wear leveling on its own flash blocks. Provided that the block-level wear leveling is effective, the problem of *channel-level wear leveling* refers to how to balance the total block erase counts of all channels.

Our design of channel-level wear leveling respects the property of *maximum parallelism* [21] for the highest parallelism among page reads. A data layout satisfies maximal parallelism if and only if a set of consecutive logical pages are mapped to the largest number of channels. This study uses the RAID-0 style striping as the initial mapping between logical pages and channels, and data updates and garbage collection do not change this mapping [17].

## 4.2 Aligning Channel Lifetime Expectancies

Provided that block wear leveling is effective, the erase counts of blocks in the same channel will be close, and the wear of a channel can be indicated by the sum of all block erase counts in this channel. Recall that the utilization of a channel stands for the fraction of host data arriving at this channel. Even though data updates are out of place at the block level, they do not change the mapping between logical pages and channels, so temporal localities have affinity with channels. Thus, channel utilizations do not abruptly change and the wear of channels increase at steady (but different) rates.

This study proposes adjusting channel utilizations to control the wear of channels for an “eventually even” state of channel lifetimes. In other words, the idea is to project channels’ lifetime expectancies to the same time point. Figure 7 is an example of two

Symbol	Description
$w$	The total amount of data written to the flash storage during $[t^-, t)$
$\bar{e}$	The write-erase cycle limit of flash blocks
$n_b$	The total number of flash blocks in a channel
$y$	The total number of channels
$C_i$	The $i$ -th channel
$e_{c_i}$	The sum of all block erase counts in the channel $C_i$
$u_{c_i}$	The utilization of the channel $C_i$ . Note that $\sum u_{c_i}=1$
$u'_{c_i}$	The expected utilization of the channel $C_i$
$r_i$	The erase ratio of the channel $C_i$
$x$	The total number of stripes
$S_i$	The $i$ -th stripe
$u_{s_i}$	The utilization of the stripe $S_i$ . Note that $\sum u_{s_i}=1$
$u_{i,j}$	The utilization of the logical block at the stripe $S_i$ and the channel $C_j$ Note that $\sum_{i=0}^{x-1} u_{i,j} = u_{c_j}$ and $\sum_{j=0}^{y-1} u_{i,j} = u_{s_i}$

Table 2: Symbol definitions.

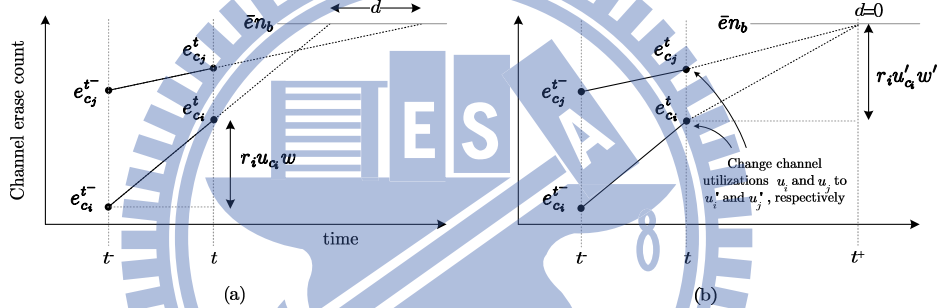


Figure 7: Aligning the lifetime expectancies of two channels  $C_i$  and  $C_j$  for channel-level wear leveling. (a) These two channels reach their end-of-life at different times. (b) Change channel utilizations  $u_{c_i}$  and  $u_{c_j}$  to  $u'_{c_i}$  and  $u'_{c_j}$ , respectively, such that the lifetime difference becomes zero (i.e.,  $d=0$ ).

channels  $C_i$  and  $C_j$ . Let every channel have the same total number of flash blocks  $n_b$ . Let a flash block endures  $\bar{e}$  write-erase cycles, and let the erase count of the channel  $C_i$ , denoted by  $e_{c_i}$ , be the sum of all block erase counts in this channel. Let a channel reach its end of life when its erase count becomes  $\bar{e} \times n_b$ . Let  $t$  be the current time, and let  $w$  be the total amount of host data written in the time interval  $[t^-, t)$ . Let  $u_{c_i} \leq 1$  be the *utilization* of the channel  $C_i$ . Thus, in this time interval the total amount of host data arriving at the channel  $C_i$  is  $u_{c_i}w$ . Let the erase counts of the channel  $C_i$  at time  $t^-$  and  $t$  be  $e_{c_i}^{t^-}$  and  $e_{c_i}^t$ , respectively. Let the *erase ratio* of  $C_i$  during  $[t^-, t)$  be  $r_i$ , defined as  $r_i = \frac{e_{c_i}^t - e_{c_i}^{t^-}}{u_{c_i}w}$ . As Fig. 7(a) shows,  $e_{c_i}$  increases by  $r_i u_{c_i} w = e_{c_i}^t - e_{c_i}^{t^-}$  in this time period. Table 2 is a summary of symbols.

Provided that channels' erase ratios and utilizations remain steady, the lifetime expectancies of the channels  $C_i$  and  $C_j$  will be  $t + (\bar{e}n_b - e_{c_i}^t) \left( \frac{t - t^-}{r_i u_{c_i} w} \right)$  and  $t + (\bar{e}n_b - e_{c_j}^t) \left( \frac{t - t^-}{r_j u_{c_j} w} \right)$ , respectively. The lifetime difference  $d$  will be

$$d = (\bar{e}n_b - e_{c_i}^t) \left( \frac{t - t^-}{r_i u_{c_i} w} \right) - (\bar{e}n_b - e_{c_j}^t) \left( \frac{t - t^-}{r_j u_{c_j} w} \right).$$

To align these two channels' lifetime expectancies (i.e.,  $d=0$ ), the channel wear-leveling algorithm computes the utilizations  $u'_{c_i}$  and  $u'_{c_j}$  which the channels  $C_i$  and  $C_j$  are expected to have after the time  $t$ , respectively. Replacing  $u_{c_i}$ ,  $u_{c_j}$ , and  $d$  in the equation above with  $u'_{c_i}$ ,  $u'_{c_j}$  and 0, respectively, produces  $u'_{c_j} = \frac{r_i(\bar{e}n_b - e_{c_j}^t)}{r_j(\bar{e}n_b - e_{c_i}^t)} u'_{c_i}$ . Because the total utilization is 100%, we have  $u'_{c_i} + u'_{c_j} = 1$ . Now solve these two equations to obtain  $u'_{c_i}$  and  $u'_{c_j}$ . Figure 7(b) shows that, with these new expected utilizations  $u'_{c_i}$  and  $u'_{c_j}$ , the lifetime expectancies of these two channels will be the same. In the general case of  $y$  channels, solving the following system obtains the expected utilizations  $u'_{c_0} \dots u'_{c_{y-1}}$ :

$$\begin{cases} \forall k \left( (k \in \{0, 1, 2, \dots, y-1\}) \wedge \left( u'_{c_k} = \frac{r_0(\bar{e}n_b - e_{c_k}^t)}{r_k(\bar{e}n_b - e_{c_0}^t)} u'_{c_0} \right) \right) \\ \sum_{k=0}^{y-1} u'_{c_k} = 1 \end{cases}.$$

The next section will present a method that swaps logical blocks among channels to adjust channel utilizations for channel wear leveling.

### 4.3 Adjusting Channel Utilizations

Independent channels adopt their own instances of flash-translation layer to manage their flash blocks. Suppose that the flash-translation layer is based on hybrid mapping. Recall that the initial mapping between logical pages and channels is the RAID-0-style striping. Let logical blocks be numbered in the channel-major order. For example, if there are four channels and a logical block is as large as four pages, then the logical block at  $lbn$  0 is in the first channel and this logical block contains the logical pages at  $lpns$  0, 4, 8, and 12. The logical block at  $lbn$  2 is in the third channel and it contains the logical pages at  $lpns$  2, 6, 10, and 14. Let a *stripe* be a set of consecutive logical blocks starting from the first channel and ending at the last channel. For example, the first stripe contains the four logical blocks at  $lbns$  0, 1, 2, and 3. Notice that these definitions of logical blocks and stripes are also applicable to page-level mapping because they are not related to space allocation in flash.

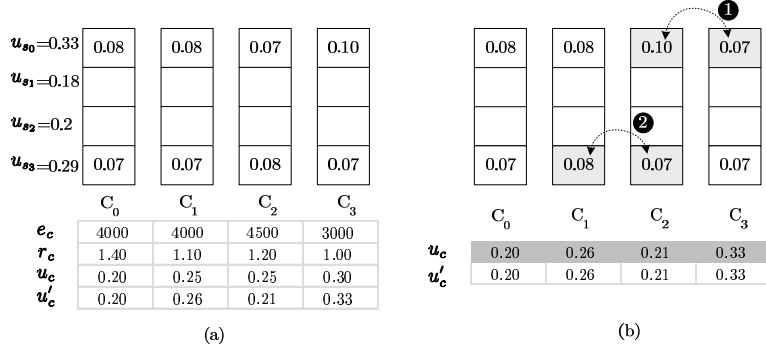


Figure 8: Swapping logical blocks among channels for channel wear leveling. (a) Before the swap and (b) after the swap.

Because real workloads have temporal localities of write, swapping logical blocks among channels can manipulate channels' future utilizations. To retain to the property of maximum parallelism, this swapping is confined to logical blocks of the same stripe. Let  $x$  be the total number of stripes. Let  $u_{s_j}$  be the utilization of the stripe  $S_j$ . Thus, we have  $\sum u_{s_j}=1$ . Let  $u_{i,j}$  be the utilization of the logical block at the stripe  $i$  and the channel  $j$ . Therefore, we have  $\sum_{i=0}^{x-1} u_{i,j} = u_{c_j}$  and  $\sum_{j=0}^{y-1} u_{i,j} = u_{s_i}$ .

This study proposes invoking channel wear leveling periodically. On each invocation, channel wear leveling computes the expected utilizations of channels, and then starts swapping logical blocks for minimizing  $\sum_{i=0}^{x-1} |u_{c_i} - u'_i|$ . This problem of block swapping is intractable even for each invocation of channel wear leveling. We can reduce any instance of the bin packing problem to this block-swapping problem. A key step of this reduction is to let an item of size  $s$  in the bin packing problem be a stripe which has only one logical block having a non-zero utilization  $s$ .

Channel wear leveling should reduce the total number of logical blocks swapped. We found that, in real workloads a stripe of a high utilization usually has two logical blocks whose utilization difference is large. This is because frequently updated data are small and they do not write to all channels [4]. Thus, the swapping begins with the stripe whose utilization is the highest. The following is a procedure to find and swap a pair of logical blocks:

**Step 1:** Find the two channels  $C_m$  and  $C_n$  which have the largest positive value of  $(u_{c_m} - u'_{c_m})$  and the smallest negative value of  $(u_{c_n} - u'_{c_n})$ , respectively.

**Step 2:** Find the stripe  $S_i$  subject to the following constraints:

- (a)  $S_i$  have the largest utilization among all stripes.
- (b) In this stripe  $S_i$ , the two logical blocks at  $C_m$  and  $C_n$  have not yet been swapped in the current invocation of channel-level wear leveling.
- (c)  $u_{i,m} > u_{i,n}$  and  $(u_{i,m} - u_{i,n}) \leq \min(u_{c_m} - u'_{c_m}, |u_{c_n} - u'_{c_n}|)$ .

**Step 3:** Exchange the channel mapping of the two logical blocks found in Step 2.

**Step 4:** Change  $u_{c_m}$  and  $u_{c_n}$  to  $(u_{c_m} - (u_{i,m} - u_{i,n}))$  and  $(u_{c_n} + (u_{i,m} - u_{i,n}))$ , respectively.

**Step 5:** Swap  $u_{i,m}$  and  $u_{i,n}$ .

In each invocation, channel wear leveling repeats Steps 1 through 5 until 1)  $u_{c_i} = u'_{c_i}$  for every  $i$  or 2) the total number of logical blocks swapped is larger than a pre-defined limitation. Figure 8 is an numeric example of channel wear leveling. In this example, the channel lifetime limit  $\bar{e}n_b$  is 10,000. Figure 8(a) shows the initial data layout and utilizations of logical blocks, channels, and stripes. Channel wear leveling solves the expected channel utilizations using  $u'_{c_3} = \frac{1.4 \times (10000 - 3000)}{1.0 \times (10000 - 4000)} = 1.63u'_{c_0}$ ,  $u'_{c_2} = 1.07u'_{c_0}$ ,  $u'_{c_1} = 1.27u'_{c_0}$ , and  $u'_{c_3} + u'_{c_2} + u'_{c_1} + u'_{c_0} = 1$ . It then selects the stripe  $S_0$  whose utilization is the highest, and swaps its two logical blocks at the channels  $C_2$  and  $C_3$ . This swap changes  $u_{c_2}$  from 0.25 to 0.22 and  $u_{c_3}$  from 0.3 to 0.33. Next, channel wear leveling selects the stripe  $S_3$  whose utilization is the second highest and swaps two more logical blocks. Figure 8(b) shows the results after these swaps. The adjusted channel utilizations match their expected utilizations.

This study proposes caching the utilization information of a small collection of most-frequently written stripes. Our experiments will show that a small cache is sufficient to effective channel wear leveling.

## 5 Performance Evaluation

### 5.1 Experimental Setup and Performance Metrics

We built a simulator and implemented two wear-leveling algorithms and Page-level mapping [11] flash-translation layer for evaluation. The simulator also implements the proposed Lazy wear leveling and Static wear leveling [8]. In addition, static wear leveling is widely used in industry [16,25].



Algorithm	largest EC	smallest EC	mean	STDDEV	Threshold	Stable
no WL	1790	0	400.31	461.89	—	no
LWL	402	69	381.51	23.23	20	<b>yes</b>
SWL	895	98	381.61	86.61	12	no

Table 3: Evaluation results of Lazy wear leveling (LWL) and Static wear leveling (SWL) under the PC workload. “no WL” stands for not using wear leveling.

Our experiments adopted PC workload. The PC workload was collected from a 40 GB hard drive in a Windows desktop for three months. The disk drive was formatted in NTFS. The user activities of this workload include web surfing, word processing, video playback, and gaming. Its write pattern consists of many temporal localities.

This study uses the standard deviation of all flash blocks’ erase counts to indicate the evenness of flash wear. The smaller the standard deviation is, the more even the flash wear will be. This study also considers the mean (i.e., the arithmetic average) of all erase counts. The difference between the means with and without wear leveling reveals the overhead of wear leveling. It is desirable for a wear-leveling algorithm to achieve a small standard deviation and a small mean.

Unless explicitly specified, all the experiments adopted the following settings as the default values: The flash page size and block size were 4KB and 512KB, respectively. This is a typical MLC-flash geometry [19]. The input workload was the PC workload, and the FTL algorithm was FAST. The *over-provisioning ratio* was 2.5%, and thus the flash size under the PC workload was  $40 \text{ GB} \times 1.025 = 41 \text{ GB}$ . Each run of the experiments replayed the input workload until 4 TB of host data were written. These replays help to differentiate the performance of different wear-leveling algorithms, but they did not manipulate the experiments.

## 5.2 Experimental Results: Block-Level Wear Leveling

This part of experiment compares Lazy wear leveling against Static wear leveling under Page-level mapping flash-translation layer. The definition of threshold on Static wear leveling is different to Lazy wear leveling. For fair comparison, this experiment fixed  $\Delta$  of Lazy wear leveling at 16, and adjusted the Static wear leveling algorithms’ thresholds to align their final erase-count means to that of Lazy wear leveling. This experiment also



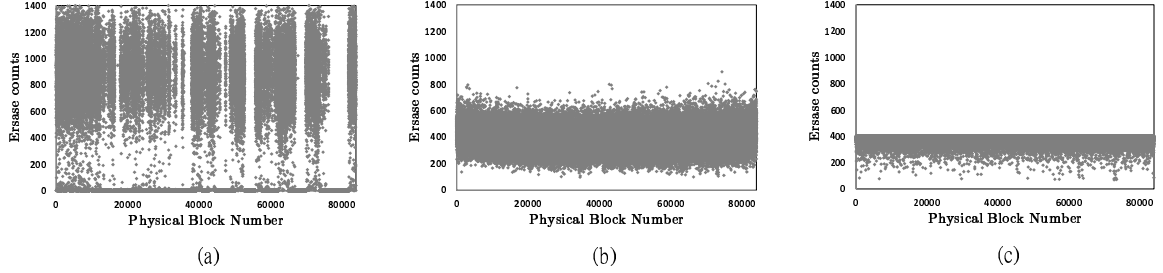


Figure 9: Final distributions of block erase counts under the PC workload. (a) Not using any wear leveling, (b) using Static wear leveling with TH=12, and (c) using Lazy wear leveling with TH=20.

adopts *stability* as a metric. Let the *stable interval* of a wear-leveling algorithm be the longest time interval  $[t_1, t_2]$  in which the standard deviations at  $t_1$  and  $t_2$  are the same. A wear-leveling algorithm is *stable* in an experiment if its stable interval length increases during the experiment. Otherwise it is *unstable*.

Table 3 shows the experimental results. First compare the results of using Static wear leveling and the results of not using wear leveling at all. The standard deviations of the PC workload is very large without wear leveling, and Lazy wear leveling reduced the standard deviation by 95% (from 461 to 23). Fig.9(a) is the erase-count distribution of experimental result of without wear leveling, some erase count of flash blocks are very high, and some are approaching 0 due to temporal locality of PC workload. Fig. 9(c) shows that, Lazy wear leveling reduced the standard deviation substantially, because Lazy wear leveling is very effective in the presence of temporal localities of write.

Next focus on the comparison among Static wear leveling and Lazy wear leveling. Lazy wear leveling outperformed Static wear leveling in terms of wear evenness in Page-level mapping. Interestingly, Static wear leveling was unstable under this workloads. Fig.9(b) shows that, under the PC workload the final erase-count distribution of Static wear leveling is more imbalanced than that of Lazy wear leveling in Fig.9(c). [6] mentioned two reasons in Hybrid mapping FTL: First, Static wear leveling moves static data from a block to another regardless of whether the target block is junior or senior. Second, Static wear leveling does not prevent the flash-translation layer from writing new data to senior blocks. Thus senior blocks could repeatedly participate in garbage collection. In contrast Lazy wear leveling neither re-maps data to a junior block nor allows the flash-translation layer to write new data to senior blocks. And these reasons are still tenable in Page-level

mapping FTL.

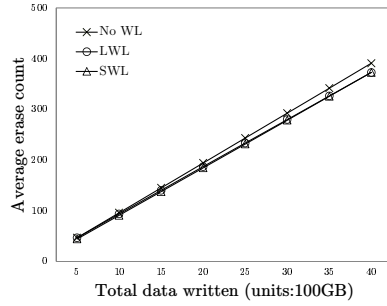


Figure 10: Average erase count on each amount of data written.

From Fig.10, we can realize that wear-leveling not only even the erase-count but also cut-down the total erase-count. This is because wear leveling performs cold data re-clustering in page-level mapping. Then increase the efficiency of garbage collection by means of reducing the number of valid pages to be copied in every garbage collection. On the other hand, it gather valid pages into same blocks, then decrease the need of garbage collection. In addition, decreasing the times of garbage collection come with reducing total erase-count of flash blocks. Therefore, total erase-count of with wear leveling will be less than without wear leveling.

### 5.3 Experimental Results: Channel-Level Wear Leveling

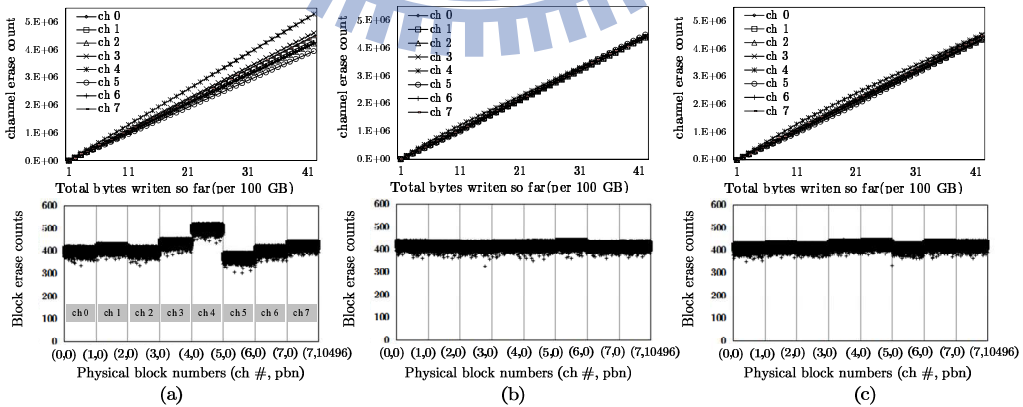


Figure 11: Runtime channel erase counts and final distributions of block erase counts under the PC workload. (a) Not using channel-level wear leveling, (b) using channel wear leveling with a period length of 512 MB, and (c) using channel wear leveling with a period length of 1 TB.

This part of experiment evaluates the proposed channel-level wear leveling algorithm. The experimental settings are as follows: The total number of channels was eight. Each channel adopts its own instance of flash-translation layer and wear-leveling algorithm. The flash-translation layer was FAST, and the wear-leveling algorithm was Lazy wear leveling. Lazy wear leveling enabled its dynamic tuning of  $\Delta$ . This experiment replayed the PC workload until 4 TB of host data were written to the flash-translation layer. The block endurance  $\bar{e}$  was set to an experimental value 450, because the average block-erase counts was near this number by the end of this experiment. The rest of the experimental settings are the same as those in Section 5.1

Figure 11 shows the runtime channel erase counts and the final distribution of block erase counts. Flash blocks are labeled by pairs (*channel number, pbn*). For example, (7,10496) refers to the last flash block in the channel number seven. The upper half of Fig. 11(a) shows that, without channel-level wear leveling, the channels 4 and 5 were the most-worn and the least-worn channels, respectively. The bottom half of 11(a) also indicates that, even though the block wear in every channel was even, the erase counts of flash blocks from the channels 4 and 5 noticeably deviated from the overall average.

Figures 11(b) and 11(c) show the results of enabling the proposed channel wear-leveling algorithm. In Fig. 11(b) and Fig. 11(c), channel wear leveling was invoked after the workload produced every 512 MB and 1TB of data, respectively, and channel wear leveling could swap up to 50 logical blocks for each invocation. The upper halves of Fig. 11(b) and 11(c) show that the channel erase counts gradually converged at the end-of-life of channels, i.e.,  $\bar{e} \times n_b = 450 \times 10,752 = 4,838,400$ . This convergence in Fig. 11(b) was faster than that in Fig. 11(c), because in Fig. 11(b) channel wear leveling was invoked more often. In the bottom halves of Fig. 11(b) and 11(c), the final distributions of block erase counts were both even. Regarding the overhead, the total numbers of logical blocks swapped were 260 and 150 when the period lengths were 512 MB and 1 TB, respectively. Compared to the overhead of serving 4 TB of host data, these overheads were almost negligible.

The proposed strategy requires a table for caching the utilization information of the most-frequently written stripes and that of their logical blocks. In this experiment, the table stored 100 stripes, and the table size was  $100 \times (1+8) \times 4 = 3,200$  bytes.

## 6 Conclusion

This study tackles two problems of wear leveling: block-level wear leveling for Page-level mapping and channel-level wear leveling. Block-level wear leveling monitors the wear of all flash blocks and intervenes when block wear develops imbalanced. Channel-level wear leveling aims at even channel lifetimes for maximizing the device-level lifespan.

This study presents two cold data selection way, and use Lazy wear leveling algorithm to copy cold data to senior block. Our results shows that Lazy wear leveling not only performs wear leveling, but also decreasing the total erase-count of flash blocks. And Lazy wear leveling for Page-level mapping is stable under real workload. It performs better on wear evenness than static wear leveling under Page-level mapping. By evening the wear and decreasing total erase-counts of flash blocks, life time of SSD will get more extension.

Multichannel architectures has become mandatory in the design of solid-state disks. Real workloads do not evenly write to all channels, and inevitably introduce imbalanced flash wear in different channels. For wear leveling at the channel level, we propose a strategy that swaps logical blocks among channels. The goal of this swapping is to reach an “eventually even” state of channel lifetimes. Results show that this strategy is very successful and its overhead is nearly negligible.

Recent study [3] suggests that SSDs in RAIDs should reach their end-of-life at different times for the convenience of drive replacement. Our future work is directed to optimizing the drive-replacement periods using the proposed lifetime projection technique.

## References

- [1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3, October 2007.
- [2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

- [3] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. *Trans. Storage*, 6(2):4:1–4:22, July 2010.
- [4] Li-Pin Chang. A hybrid approach to nand-flash-based solid-state disks. *Computers, IEEE Transactions on*, 59(10):1337–1349, oct. 2010.
- [5] Li-Pin Chang and Chun-Da Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1):1–36, 2009.
- [6] Li-Pin Chang and Li-Chun Huang. A low-cost wear-leveling algorithm for block-mapping solid-state disks. In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pages 31–40, New York, NY, USA, 2011. ACM.
- [7] Li-Pin Chang and Tei-Wei Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196, 2002.
- [8] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Transactions on Computers*, 59(1):53–65, jan. 2010.
- [9] Cagdas Dirik and Bruce Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.
- [10] Global Unichip Corp. GP5086 Datasheet. <http://www.globalunichip.com/4-10.php>, 2009.
- [11] Aayush Gupta, Youngjae Kim, and Bhuvan Uргаonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240. ACM, 2009.

- [12] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164. ACM, 2007.
- [13] Jeong-Uk Kang, Jin-Soo Kim, Chanik Park, Hyoungjun Park, and Joonwon Lee. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.*, 53(9):644–658, 2007.
- [14] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*, 6(3):18, 2007.
- [15] Micron<sup>®</sup>. *Wear-Leveling Techniques in NAND Flash Devices*. Micron Application Note (TN-29-42), 2008.
- [16] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl architecture for nand flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–23, 2008.
- [17] Seung-Ho Park, Jung-Wook Park, Shin-Dug Kim, and Charles C. Weems. A pattern adaptive nand flash memory storage structure. *IEEE Transactions on Computers*, 99(PrePrints), 2010.
- [18] Samsung Electronics. *K9F8G08B0M 1Gb \* 8 Bit SLC NAND Flash Memory*. Data sheet, 2006.
- [19] Samsung Electronics. *K9MDG08U5M 4G \* 8 Bit MLC NAND Flash Memory*. Data sheet, 2008.
- [20] Yoon Jae Seong, Eyeeye Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin-Yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, Yookun Cho, and Sang Lyul Min. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Transactions on Computers*, 59:905–921, 2010.
- [21] Pengiu Shang, Jun Wang, Huijun Zhu, and Peng Gu. A new placement-ideal layout for multiway replication storage system. *Computers, IEEE Transactions on*, 60(8):1142–1156, aug. 2011.

[22] Spansion<sup>®</sup>. *Wear Leveling*. Spansion Application Note (AN01), 2008.

